

Statistik und ihre Anwendungen

Daniel Wollschläger

Grundlagen der Datenanalyse mit R

Eine anwendungsorientierte Einführung

3. Auflage



Springer Spektrum

Statistik und ihre Anwendungen

Herausgegeben von

H. Dette, Bochum, Deutschland
W. K. Härdle, Berlin, Deutschland

Herausgegeben von

Holger Dette
Univ. Bochum Inst. Mathematik
Bochum, Deutschland

Wolfgang Karl Härdle
Berlin, Deutschland

Daniel Wollschläger

Grundlagen der Datenanalyse mit R

Eine anwendungsorientierte Einführung

3., überarbeitete und erweiterte Auflage

Daniel Wollschläger
Institut für Medizinische Biometrie,
Epidemiologie und Informatik (IMBEI)
Universitätsmedizin der Johannes
Gutenberg-Universität Mainz
Mainz, Deutschland

Statistik und ihre Anwendungen
ISBN 978-3-662-45506-7 ISBN 978-3-662-45507-4 (eBook)
DOI 10.1007/978-3-662-45507-4

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Springer Spektrum
© Springer-Verlag Berlin Heidelberg 2010, 2012, 2014
Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften. Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen.

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Berlin Heidelberg ist Teil der Fachverlagsgruppe Springer Science+Business Media
(www.springer.com)

Vorwort

Das vorliegende Buch liefert eine an human- und sozialwissenschaftlichen Anwendungen orientierte Einführung in die Datenauswertung mit R. R ist eine freie Umgebung zur umfassenden statistischen Analyse und grafischen Darstellung von Datensätzen, die befehlsorientiert arbeitet. Dieser Text soll jenen den Einstieg in R erleichtern, die in erster Linie grundlegende Auswertungsverfahren anwenden möchten und über keine Vorkenntnisse mit Programmen ohne grafische Benutzeroberfläche verfügen.

Die hier getroffene Auswahl an statistischen Verfahren orientiert sich an den Anforderungen der Psychologie, soll aber auch die wichtigsten Auswertungsmethoden anderer Human- und Sozialwissenschaften sowie der klinischen Forschung abdecken. Das Buch stellt die Umsetzung grafischer und deskriptiver Datenauswertung, nonparametrischer Verfahren, univariater linearer Modelle und multivariater Methoden vor. Dabei liegt der Fokus auf der Umsetzung der Verfahren mit R, nicht aber auf der Vermittlung statistischer Grundlagen. Von diesen wird hier angenommen, dass die Leser mit ihnen vertraut sind. Auf Literatur zu den behandelten Verfahren wird zu Beginn der Abschnitte jeweils hingewiesen.

Kapitel 1 bis 3 dienen der Einführung in den Umgang mit R, in die zur Datenanalyse notwendigen Grundkonzepte sowie in die Syntax der Befehlssteuerung. Inhaltlich werden in Kapitel 2 Methoden zur deskriptiven Datenauswertung behandelt, Kapitel 3 befasst sich mit der Organisation von Datensätzen. Das sich an Kapitel 4 zur Verwaltung von Befehlen und Daten anschließende Kapitel 5 stellt Hilfsmittel für die inferenzstatistischen Methoden bereit. Diese werden in Kapitel 6 (lineare Regression), 7 (*t*-Tests und Varianzanalysen), 8 (Regression für kategoriale Daten), 9 (Survival-Analyse), 10 (klassische nonparametrische Tests), 11 (bootstrap und Permutationstests), 12 (multivariate Methoden) und 13 (Vorhersagegüte prädiktiver Modelle) behandelt. Das Buch schließt mit Kapitel 14 zum Erstellen von Diagrammen und einem Ausblick auf den Einsatz von R als Programmiersprache in Kapitel 15. Diese Reihenfolge der Themen ist bei der Lektüre keinesfalls zwingend einzuhalten. Da statistische Analysen meist gemeinsam mit der Datenorganisation und grafischen Illustration durchzuführen sind, empfiehlt es sich vielmehr, bereits zu Beginn auch Kapitel 4 und 14 selektiv parallel zu lesen.

Um die Ergebnisse der R-eigenen Auswertungsfunktionen besser nachvollziehbar zu machen, wird ihre Anwendung an vielen Stellen durch manuelle Kontrollrechnungen begleitet. Die eigene Umsetzung soll zudem zeigen, wie auch Testverfahren, für die zunächst keine vorbereitete Funktion vorhanden ist, mit elementaren Mitteln prinzipiell selbst umgesetzt werden können. In den meisten Beispielen wird davon ausgegangen, dass die vorliegenden Daten bereits geprüft sind und eine hohe Datenqualität vorliegt: Fragen der Einheitlichkeit etwa hinsichtlich der Codierung von Datum und Uhrzeit, potentiell unvollständige Datensätze, fehlerhaft eingegebene oder unplausible Daten sowie doppelte Werte oder Ausreißer sollen ausgeschlossen sein. Besondere Aufmerksamkeit wird jedoch in einem eigenen Abschnitt dem Thema fehlender Werte geschenkt.

Änderungen in der zweiten Auflage

Gegenüber der vorangehenden Auflage wurde das Buch stärker aufgabenorientiert strukturiert sowie an vielen Stellen überarbeitet und inhaltlich ergänzt. So geht es etwa ausführlicher auf die Verarbeitung von Zeichenketten und Datumsangaben ein (Abschn. 2.12, 2.13) und beinhaltet eine vertiefte Darstellung der Diagnostik und Kreuzvalidierung von Regressionsmodellen (Abschn. 6.5, 13.1). Die Auswertung varianzanalytischer Fragestellungen berücksichtigt jetzt die Schätzung von Effektstärken (Abschn. 7.2–7.7). Als Tests auf gleiche Variabilität werden zusätzlich jene nach Fligner-Killeen sowie nach Mood und Ansari-Bradley besprochen (7.1.3, 10.4). Das neue Kapitel 11 führt in die Anwendung von bootstrapping und Permutationstests ein. Bei multivariaten Verfahren ist die Diskriminanzanalyse ebenso hinzugekommen wie eine Behandlung des allgemeinen linearen Modells (Abschn. 12.8, 12.9). Schließlich geht der Text nun auf Möglichkeiten zur Darstellung von Bitmap-Grafiken ein (Abschn. 14.5.10) und beschreibt detaillierter, welche Möglichkeiten für Funktionsanalyse und debugging R bietet (Abschn. 15.3). Die R-Beispieldaten sind ausführlicher kommentiert und abschnittsübergreifend konsistenter. Der überarbeitete Index wurde nach inhaltlichen Schlagworten, R-Funktionen und Zusatzpaketen getrennt.

Änderungen in der dritten Auflage

In den vergangenen Jahren haben sich R und insbesondere die mit R verknüpfte Software-Landschaft stark weiterentwickelt. Hervorzuheben ist hier zum einen die Verfügbarkeit grafischer Entwicklungsumgebungen, die weit komfortabler und funktionsreicher als die Oberfläche der R-Basisinstallation sind (vgl. Abschnitt 1.1.4).

Zum anderen ist die Anzahl von Zusatzpaketen weiter exponentiell gewachsen. Mittlerweile sind viele Zusatzpaket auch für Aufgaben beliebt, die sich mit Mitteln des Basisumfangs zwar lösen lassen, dies aber komplizierter oder weniger elegant ist. Dies gilt insbesondere für den Umgang mit Zeichenketten (vgl. Abschn. 2.12) und mit Datumsangaben (vgl. Abschn. 2.13), für die Transformation von Datensätzen (vgl. Abschn. 3.3, 3.4) sowie für Alternativen zu den Basis-Diagrammen (vgl. Kap. 14, Abschn. 14.8.3). In diesem Buch finden sich viele Hinweise auf Erweiterungsmöglichkeiten mit Zusatzpaketen. Die Darstellung konzentriert sich aber auf den Basisumfang von R, der eine reife und stabile Grundlage für sich derzeit häufig noch dynamisch ändernde Erweiterungen darstellt.

In der vorliegenden dritten Auflage bezieht sich das Buch auf Version 3.1.2 von R. Neben vielen Detailänderungen wurde die Auswahl verwendeter Zusatzpaket den neuen Entwicklungen angepasst.

Abschnitt 4.2.4 behandelt ausführlicher den Datenaustausch mit Datenbanken. Der neue Abschnitt 4.3 stellt vor, wie man mit Dateien und Pfaden arbeitet. Hinweise auf Erweiterungen der linearen Regression liefert Abschn. 6.6 – etwa auf robuste, penalisierte oder gemischte Modelle sowie auf verallgemeinerte Schätzgleichungen. Regressionsmodelle für kategoriale Daten und Zähldaten sind nun in Kap. 8 zusammengefasst und wurden um die ordinale (Abschn. 8.2), multinomiale (Abschn. 8.3) und Poisson-Regression (Abschn. 8.4) ergänzt. Abschnitt 8.5 beschreibt log-lineare Modelle. Das neue Kapitel 9 führt in die Analyse von Ereigniszeiten ein (Kaplan-Meier in Abschn. 9.3, Cox proportional hazards in Abschn. 9.4 und parametrische Modelle in Abschn. 9.5). ROC-Kurven und AUC werden nun in Abschn. 10.2.7 beschrieben. Abschnitt 11.1.3 zeigt ein Beispiel für stratifiziertes bootstrapping, Abschn. 11.1.6 demonstriert den wild bootstrap für lineare Modelle. Der Abschnitt zur Kreuzvalidierung linearer Modelle

wurde zu Kap. 13 erweitert, das auch die Vorhersagegüte in verallgemeinerten linearen Modellen behandelt sowie Bootstrap-Methoden zur unverzerrten Schätzung des Vorhersagefehlers vorstellt. Wie man mit dem Paket `ggplot2` Diagramme erstellt, erläutert in Grundzügen Abschn. 14.8.3.

Korrekturen, Ergänzungen und Anregungen sind herzlich willkommen. Die verwendeten Daten sowie alle Befehle des Buches und ggf. notwendige Berichtigungen erhalten Sie online:

<http://www.dwoll.de/r/>

Danksagung

Mein besonderer Dank gilt den Personen, die an der Entstehung des Buches in frühen und späteren Phasen mitgewirkt haben: Abschn. 1.1 bis 1.2.3 entstanden auf der Grundlage eines Manuskripts von Dieter Heyer und Gisela Müller-Plath am Institut für Psychologie der Martin-Luther-Universität Halle-Wittenberg, denen ich für die Überlassung des Textes danken möchte. Zahlreiche Korrekturen und viele Verbesserungsvorschläge wurden dankenswerterweise von Andri Signorell, Ulrike Groemping, Wolfgang Ramos, Julian Etzel, Erwin Grüner, Johannes Andres, Sabrina Flindt und Susanne Wollschläger beigesteuert. Johannes Andres danke ich für seine ausführlichen Erläuterungen der statistischen Grundlagen. Die Entstehung des Buches wurde beständig durch die selbstlose Unterstützung von Heike Jores, Martha Jores und Vincent van Houten begleitet. Clemens Heine, Niels Peter Thomas und Alice Blanck vom Springer Verlag möchte ich herzlich für die freundliche Kooperation und Begleitung der Veröffentlichung danken.

Zuvorderst ist aber dem Entwickler-Team von R sowie der zahlreichen Zusatzpakete Dank und Anerkennung dafür zu zollen, dass sie in freiwillig geleisteter Arbeit eine hervorragende Umgebung zur statistischen Datenauswertung geschaffen haben, deren mächtige Funktionalität hier nur zu einem Bruchteil vorgestellt werden kann.

Mainz,
Oktober 2014

Daniel Wollschläger
`contact@dwoll.de`

Inhaltsverzeichnis

1 Erste Schritte	1
1.1 Vorstellung	1
1.1.1 Pro und Contra R	1
1.1.2 Typografische Konventionen	3
1.1.3 R installieren	3
1.1.4 Grafische Benutzeroberflächen	4
1.1.5 Weiterführende Informationsquellen und Literatur	5
1.2 Grundlegende Elemente	6
1.2.1 R Starten, beenden und die Konsole verwenden	6
1.2.2 Einstellungen	10
1.2.3 Umgang mit dem workspace	11
1.2.4 Einfache Arithmetik	13
1.2.5 Funktionen mit Argumenten aufrufen	15
1.2.6 Hilfe-Funktionen	16
1.2.7 Zusatzpakete verwenden	16
1.2.8 Empfehlungen und typische Fehlerquellen	18
1.3 Datenstrukturen: Klassen, Objekte, Datentypen	19
1.3.1 Objekte benennen	20
1.3.2 Zuweisungen an Objekte	21
1.3.3 Objekte ausgeben	21
1.3.4 Objekte anzeigen lassen, umbenennen und entfernen	22
1.3.5 Datentypen	23
1.3.6 Logische Werte, Operatoren und Verknüpfungen	24
2 Elementare Dateneingabe und -verarbeitung	27
2.1 Vektoren	27
2.1.1 Vektoren erzeugen	27
2.1.2 Elemente auswählen und verändern	28
2.1.3 Datentypen in Vektoren	31
2.1.4 Elemente benennen	31
2.1.5 Elemente löschen	32
2.2 Logische Operatoren	32
2.2.1 Vektoren mit logischen Operatoren vergleichen	33
2.2.2 Logische Indexvektoren	35
2.3 Mengen	37
2.3.1 Doppelt auftretende Werte finden	37
2.3.2 Mengenoperationen	37
2.3.3 Kombinatorik	39

Inhaltsverzeichnis

2.4	Systematische und zufällige Wertefolgen erzeugen	41
2.4.1	Numerische Sequenzen erstellen	42
2.4.2	Wertefolgen wiederholen	43
2.4.3	Zufällig aus einer Urne ziehen	43
2.4.4	Zufallszahlen aus bestimmten Verteilungen erzeugen	44
2.5	Daten transformieren	45
2.5.1	Werte sortieren	45
2.5.2	Werte in zufällige Reihenfolge bringen	46
2.5.3	Teilmengen von Daten auswählen	47
2.5.4	Daten umrechnen	48
2.5.5	Neue aus bestehenden Variablen bilden	51
2.5.6	Werte ersetzen oder recodieren	51
2.5.7	Kontinuierliche Variablen in Kategorien einteilen	53
2.6	Gruppierungsfaktoren	54
2.6.1	Ungeordnete Faktoren	54
2.6.2	Faktoren kombinieren	56
2.6.3	Faktorstufen nachträglich ändern	57
2.6.4	Geordnete Faktoren	59
2.6.5	Reihenfolge von Faktorstufen bestimmen	59
2.6.6	Faktoren nach Muster erstellen	61
2.6.7	Quantitative in kategoriale Variablen umwandeln	62
2.7	Deskriptive Kennwerte numerischer Daten	63
2.7.1	Summen, Differenzen und Produkte	63
2.7.2	Extremwerte	64
2.7.3	Mittelwert, Median und Modalwert	65
2.7.4	Robuste Maße der zentralen Tendenz	67
2.7.5	Prozentrang, Quartile und Quantile	68
2.7.6	Varianz, Streuung, Schiefe und Wölbung	69
2.7.7	Diversität kategorialer Daten	70
2.7.8	Kovarianz und Korrelation	70
2.7.9	Robuste Streuungsmaße und Kovarianzschatzter	72
2.7.10	Kennwerte getrennt nach Gruppen berechnen	73
2.7.11	Funktionen auf geordnete Paare von Werten anwenden	75
2.8	Matrizen	75
2.8.1	Datentypen in Matrizen	76
2.8.2	Dimensionierung, Zeilen und Spalten	76
2.8.3	Elemente auswählen und verändern	78
2.8.4	Weitere Wege, Elemente auszuwählen und zu verändern	80
2.8.5	Matrizen verbinden	81
2.8.6	Matrizen sortieren	82
2.8.7	Randkennwerte berechnen	83
2.8.8	Beliebige Funktionen auf Matrizen anwenden	83
2.8.9	Matrix zeilen- oder spaltenweise mit Kennwerten verrechnen	84
2.8.10	Kovarianz- und Korrelationsmatrizen	85
2.9	Arrays	86
2.10	Häufigkeitsauszählungen	88
2.10.1	Einfache Tabellen absoluter und relativer Häufigkeiten	88

Inhaltsverzeichnis

2.10.2 Iterationen zählen	90
2.10.3 Absolute, relative und bedingte relative Häufigkeiten in Kreuztabellen .	90
2.10.4 Randkennwerte von Kreuztabellen	94
2.10.5 Datensätze aus Häufigkeitstabellen erstellen	94
2.10.6 Kumulierte relative Häufigkeiten und Prozentrang	95
2.11 Fehlende Werte behandeln	96
2.11.1 Fehlende Werte codieren und identifizieren	97
2.11.2 Fehlende Werte ersetzen und umcodieren	98
2.11.3 Behandlung fehlender Werte bei der Berechnung einfacher Kennwerte .	99
2.11.4 Behandlung fehlender Werte in Matrizen	100
2.11.5 Behandlung fehlender Werte beim Sortieren von Daten	102
2.11.6 Behandlung fehlender Werte in inferenzstatistischen Tests	102
2.11.7 Multiple Imputation	103
2.12 Zeichenketten verarbeiten	103
2.12.1 Objekte in Zeichenketten umwandeln	103
2.12.2 Zeichenketten erstellen und ausgeben	104
2.12.3 Zeichenketten manipulieren	107
2.12.4 Zeichenfolgen finden	108
2.12.5 Zeichenfolgen extrahieren	109
2.12.6 Zeichenfolgen ersetzen	110
2.12.7 Zeichenketten als Befehl ausführen	111
2.13 Datum und Uhrzeit	112
2.13.1 Datumsangaben erstellen und formatieren	112
2.13.2 Uhrzeit	113
2.13.3 Mit Datum und Uhrzeit rechnen	115
3 Datensätze	117
3.1 Listen	117
3.1.1 Komponenten auswählen und verändern	118
3.1.2 Komponenten hinzufügen und entfernen	120
3.1.3 Listen mit mehreren Ebenen	121
3.2 Datensätze	122
3.2.1 Datentypen in Datensätzen	124
3.2.2 Elemente auswählen und verändern	125
3.2.3 Namen von Variablen und Beobachtungen	127
3.2.4 Datensätze in den Suchpfad einfügen	128
3.3 Datensätze transformieren	129
3.3.1 Variablen hinzufügen und entfernen	130
3.3.2 Datensätze sortieren	131
3.3.3 Teilmengen von Daten mit <code>subset()</code> auswählen	132
3.3.4 Doppelte und fehlende Werte behandeln	135
3.3.5 Datensätze teilen	136
3.3.6 Datensätze zeilen- oder spaltenweise verbinden	137
3.3.7 Datensätze mit <code>merge()</code> zusammenführen	138
3.3.8 Organisationsform einfacher Datensätze ändern	141
3.3.9 Organisationsform komplexer Datensätze ändern	143

Inhaltsverzeichnis

3.4	Daten aggregieren	147
3.4.1	Funktionen auf Variablen anwenden	147
3.4.2	Funktionen für mehrere Variablen anwenden	150
3.4.3	Funktionen getrennt nach Gruppen anwenden	151
4	Befehle und Daten verwalten	153
4.1	Befehlssequenzen im Editor bearbeiten	153
4.2	Daten importieren und exportieren	154
4.2.1	Daten im Editor eingeben	155
4.2.2	Datentabellen im Textformat	155
4.2.3	R-Objekte	158
4.2.4	Daten mit anderen Programmen austauschen	158
4.2.5	Daten in der Konsole einlesen	165
4.2.6	Unstrukturierte Textdateien	165
4.2.7	Datenqualität sicherstellen	166
4.3	Dateien verwalten	167
4.3.1	Dateien auswählen	167
4.3.2	Dateipfade manipulieren	168
4.3.3	Dateien verändern	169
5	Hilfsmittel für die Inferenzstatistik	171
5.1	Wichtige Begriffe inferenzstatistischer Tests	171
5.2	Lineare Modelle formulieren	172
5.3	Funktionen von Zufallsvariablen	174
5.3.1	Dichtefunktion	175
5.3.2	Verteilungsfunktion	176
5.3.3	Quantilfunktion	177
6	Lineare Regression	179
6.1	Test auf Korrelation	179
6.2	Einfache lineare Regression	180
6.2.1	Deskriptive Modellanpassung	181
6.2.2	Regressionsanalyse	183
6.3	Multiple lineare Regression	186
6.3.1	Deskriptive Modellanpassung und Regressionsanalyse	186
6.3.2	Modell verändern	188
6.3.3	Modelle vergleichen und auswählen	189
6.3.4	Moderierte Regression	191
6.4	Regressionsmodelle auf andere Daten anwenden	194
6.5	Regressionsdiagnostik	195
6.5.1	Extremwerte, Ausreißer und Einfluss	196
6.5.2	Verteilungseigenschaften der Residuen	199
6.5.3	Multikollinearität	201
6.6	Erweiterungen der linearen Regression	203
6.6.1	Robuste Regression	203
6.6.2	Penalisierte Regression	204
6.6.3	Nichtlineare Zusammenhänge	207

6.6.4	Abhängige Fehler bei Messwiederholung oder Clusterung	208
6.7	Partialkorrelation und Semipartialkorrelation	208
7	t-Tests und Varianzanalysen	212
7.1	Tests auf Varianzhomogenität	212
7.1.1	<i>F</i> -Test auf Varianzhomogenität für zwei Stichproben	212
7.1.2	Levene-Test für mehr als zwei Stichproben	213
7.1.3	Fligner-Killeen-Test für mehr als zwei Stichproben	214
7.2	<i>t</i> -Tests	215
7.2.1	<i>t</i> -Test für eine Stichprobe	215
7.2.2	<i>t</i> -Test für zwei unabhängige Stichproben	217
7.2.3	<i>t</i> -Test für zwei abhängige Stichproben	219
7.3	Einfaktorielle Varianzanalyse (CR- <i>p</i>)	220
7.3.1	Auswertung mit <code>oneway.test()</code>	220
7.3.2	Auswertung mit <code>aov()</code>	221
7.3.3	Auswertung mit <code>anova()</code>	223
7.3.4	Effektstärke schätzen	223
7.3.5	Voraussetzungen grafisch prüfen	224
7.3.6	Einzelvergleiche (Kontraste)	225
7.4	Einfaktorielle Varianzanalyse mit abhängigen Gruppen (RB- <i>p</i>)	231
7.4.1	Univariat formuliert auswerten und Effektstärke schätzen	232
7.4.2	Zirkularität der Kovarianzmatrix prüfen	235
7.4.3	Multivariat formuliert auswerten mit <code>Anova()</code>	237
7.4.4	Multivariat formuliert auswerten mit <code>anova()</code>	238
7.4.5	Einzelvergleiche und alternative Auswertungsmöglichkeiten	239
7.5	Zweifaktorielle Varianzanalyse (CRF- <i>pq</i>)	239
7.5.1	Auswertung und Schätzung der Effektstärke	240
7.5.2	Quadratsummen vom Typ I, II und III	242
7.5.3	Bedingte Haupteffekte testen	246
7.5.4	Beliebige a-priori Kontraste	249
7.5.5	Beliebige post-hoc Kontraste nach Scheffé	252
7.5.6	Marginale Paarvergleiche nach Tukey	253
7.6	Zweifaktorielle Varianzanalyse mit zwei Intra-Gruppen Faktoren (RBF- <i>pq</i>)	254
7.6.1	Univariat formuliert auswerten und Effektstärke schätzen	254
7.6.2	Zirkularität der Kovarianzmatrizen prüfen	258
7.6.3	Multivariat formuliert auswerten	259
7.6.4	Einzelvergleiche (Kontraste)	260
7.7	Zweifaktorielle Varianzanalyse mit Split-Plot-Design (SPF- <i>p · q</i>)	261
7.7.1	Univariat formuliert auswerten und Effektstärke schätzen	261
7.7.2	Voraussetzungen und Prüfen der Zirkularität	264
7.7.3	Multivariat formuliert auswerten	265
7.7.4	Einzelvergleiche (Kontraste)	266
7.7.5	Erweiterung auf dreifaktorielles SPF- <i>p · qr</i> Design	267
7.7.6	Erweiterung auf dreifaktorielles SPF- <i>pq · r</i> Design	269
7.8	Kovarianzanalyse	270
7.8.1	Test der Effekte von Gruppenzugehörigkeit und Kovariate	270
7.8.2	Beliebige a-priori Kontraste	276

Inhaltsverzeichnis

7.8.3	Beliebige post-hoc Kontraste nach Scheffé	277
7.9	Power, Effektstärke und notwendige Stichprobengröße	278
7.9.1	Binomialtest	278
7.9.2	<i>t</i> -Test	280
7.9.3	Einfaktorielle Varianzanalyse	283
8	Regressionsmodelle für kategoriale Daten und Zähldaten	287
8.1	Logistische Regression	288
8.1.1	Modell für dichotome Daten anpassen	288
8.1.2	Modell für binomiale Daten anpassen	290
8.1.3	Anpassungsgüte	291
8.1.4	Vorhersage, Klassifikation und Anwendung auf neue Daten	293
8.1.5	Signifikanztests für Parameter und Modell	295
8.1.6	Andere Link-Funktionen	297
8.1.7	Mögliche Probleme bei der Modellanpassung	297
8.2	Ordinale Regression	298
8.2.1	Modellanpassung	299
8.2.2	Anpassungsgüte	300
8.2.3	Signifikanztests für Parameter und Modell	301
8.2.4	Vorhersage, Klassifikation und Anwendung auf neue Daten	303
8.3	Multinomiale Regression	304
8.3.1	Modellanpassung	305
8.3.2	Anpassungsgüte	306
8.3.3	Signifikanztests für Parameter und Modell	307
8.3.4	Vorhersage, Klassifikation und Anwendung auf neue Daten	308
8.4	Regression für Zähldaten	309
8.4.1	Poisson-Regression	309
8.4.2	Ereignisraten analysieren	311
8.4.3	Adjustierte Poisson-Regression und negative Binomial-Regression	312
8.4.4	Zero-inflated Poisson-Regression	314
8.4.5	Zero-truncated Poisson-Regression	317
8.5	Log-lineare Modelle	317
8.5.1	Modell	317
8.5.2	Modellanpassung	319
9	Survival-Analyse	324
9.1	Verteilung von Ereignisz Zeiten	324
9.2	Zensierte und gestützte Ereignisz Zeiten	325
9.2.1	Zeitlich konstante Prädiktoren	326
9.2.2	Daten in Zählprozess-Darstellung	328
9.3	Kaplan-Meier-Analyse	331
9.3.1	Survival-Funktion und kumulatives hazard schätzen	331
9.3.2	Log-Rank-Test auf gleiche Survival-Funktionen	333
9.4	Cox proportional hazards Modell	334
9.4.1	Anpassungsgüte und Modelltests	337
9.4.2	Survival-Funktion und baseline hazard schätzen	338
9.4.3	Modelldiagnostik	340

Inhaltsverzeichnis

9.4.4 Vorhersage und Anwendung auf neue Daten	344
9.4.5 Erweiterungen des Cox PH-Modells	345
9.5 Parametrische proportional hazards Modelle	345
9.5.1 Darstellung über die Hazard-Funktion	345
9.5.2 Darstellung als accelerated failure time Modell	346
9.5.3 Anpassung und Modelltests	347
9.5.4 Survival-Funktion schätzen	348
10 Klassische nonparametrische Methoden	351
10.1 Anpassungstests	351
10.1.1 Binomialtest	352
10.1.2 Test auf Zufälligkeit (Runs-Test)	353
10.1.3 Kolmogorov-Smirnov-Anpassungstest	355
10.1.4 χ^2 -Test auf eine feste Verteilung	358
10.1.5 χ^2 -Test auf eine Verteilungsklasse	359
10.2 Analyse von gemeinsamen Häufigkeiten kategorialer Variablen	361
10.2.1 χ^2 -Test auf Unabhängigkeit	361
10.2.2 χ^2 -Test auf Gleichheit von Verteilungen	362
10.2.3 χ^2 -Test für mehrere Auftretenswahrscheinlichkeiten	363
10.2.4 Fishers exakter Test auf Unabhängigkeit	364
10.2.5 Fishers exakter Test auf Gleichheit von Verteilungen	365
10.2.6 Kennwerte von (2×2) -Konfusionsmatrizen	366
10.2.7 ROC-Kurve und AUC	369
10.3 Maße für Zusammenhang und Übereinstimmung	371
10.3.1 Spearmans ρ und Kendalls τ	371
10.3.2 Zusammenhang kategorialer Variablen	373
10.3.3 Inter-Rater-Übereinstimmung	374
10.4 Tests auf gleiche Variabilität	382
10.4.1 Mood-Test	382
10.4.2 Ansari-Bradley-Test	383
10.5 Tests auf Übereinstimmung von Verteilungen	384
10.5.1 Kolmogorov-Smirnov-Test für zwei Stichproben	384
10.5.2 Vorzeichen-Test	386
10.5.3 Wilcoxon-Vorzeichen-Rang-Test für eine Stichprobe	387
10.5.4 Wilcoxon-Rangsummen-Test / Mann-Whitney-U-Test	389
10.5.5 Wilcoxon-Test für zwei abhängige Stichproben	390
10.5.6 Kruskal-Wallis-H-Test für unabhängige Stichproben	390
10.5.7 Friedman-Rangsummen-Test für abhängige Stichproben	392
10.5.8 Cochran-Q-Test für abhängige Stichproben	394
10.5.9 Bowker-Test für zwei abhängige Stichproben	395
10.5.10 McNemar-Test für zwei abhängige Stichproben	396
10.5.11 Stuart-Maxwell-Test für zwei abhängige Stichproben	398
11 Resampling-Verfahren	400
11.1 Bootstrapping	400
11.1.1 Replikationen erstellen	401
11.1.2 Bootstrap-Vertrauensintervalle für μ	404

Inhaltsverzeichnis

11.1.3	Bootstrap-Vertrauensintervalle für $\mu_2 - \mu_1$	406
11.1.4	Lineare Modelle: case resampling	407
11.1.5	Lineare Modelle: model-based resampling	409
11.1.6	Lineare Modelle: wild bootstrap	411
11.2	Permutationstests	412
11.2.1	Test auf gleiche Lageparameter in unabhängigen Stichproben	413
11.2.2	Test auf gleiche Lageparameter in abhängigen Stichproben	415
11.2.3	Test auf Unabhängigkeit von zwei Variablen	416
12	Multivariate Verfahren	418
12.1	Lineare Algebra	418
12.1.1	Matrix-Algebra	418
12.1.2	Lineare Gleichungssysteme lösen	422
12.1.3	Norm und Abstand von Vektoren und Matrizen	422
12.1.4	Mahalanobistransformation und Mahalanobisdistanz	424
12.1.5	Kennwerte von Matrizen	426
12.1.6	Zerlegungen von Matrizen	428
12.1.7	Orthogonale Projektion	430
12.2	Hauptkomponentenanalyse	433
12.2.1	Berechnung	434
12.2.2	Dimensionsreduktion	437
12.3	Faktorenanalyse	439
12.4	Multidimensionale Skalierung	446
12.5	Multivariate multiple Regression	447
12.6	Hotellings T^2	449
12.6.1	Test für eine Stichprobe	449
12.6.2	Test für zwei unabhängige Stichproben	451
12.6.3	Test für zwei abhängige Stichproben	453
12.6.4	Univariate Varianzanalyse mit abhängigen Gruppen (RB- p)	454
12.7	Multivariate Varianzanalyse (MANOVA)	455
12.7.1	Einfaktorielle MANOVA	455
12.7.2	Zweifaktorielle MANOVA	456
12.8	Diskriminanzanalyse	457
12.9	Das allgemeine lineare Modell	462
12.9.1	Modell der multiplen linearen Regression	462
12.9.2	Modell der einfaktoriellen Varianzanalyse	464
12.9.3	Modell der zweifaktoriellen Varianzanalyse	469
12.9.4	Parameterschätzungen, Vorhersage und Residuen	473
12.9.5	Hypothesen über parametrische Funktionen testen	475
12.9.6	Lineare Hypothesen als Modellvergleiche formulieren	475
12.9.7	Lineare Hypothesen testen	480
12.9.8	Beispiel: Multivariate multiple Regression	483
12.9.9	Beispiel: Einfaktorielle MANOVA	485
12.9.10	Beispiel: Zweifaktorielle MANOVA	488

13 Vorhersagegüte prädiktiver Modelle	492
13.1 Kreuzvalidierung linearer Regressionsmodelle	492
13.1.1 k -fache Kreuzvalidierung	493
13.1.2 Leave-One-Out Kreuzvalidierung	494
13.2 Kreuzvalidierung verallgemeinerter linearer Modelle	495
13.3 Bootstrap-Vorhersagefehler	496
14 Diagramme erstellen	499
14.1 Grafik-Devices	499
14.1.1 Aufbau und Verwaltung von Grafik-Devices	499
14.1.2 Grafiken speichern	501
14.2 Streu- und Liniendiagramme	502
14.2.1 Streudiagramme mit <code>plot()</code>	502
14.2.2 Datenpunkte eines Streudiagramms identifizieren	504
14.2.3 Streudiagramme mit <code>matplotlib</code>	505
14.3 Diagramme formatieren	505
14.3.1 Grafikelemente formatieren	505
14.3.2 Farben spezifizieren	508
14.3.3 Achsen formatieren	511
14.4 Säulen- und Punktdiagramme	511
14.4.1 Einfache Säulendiagramme	512
14.4.2 Gruppierte und gestapelte Säulendiagramme	512
14.4.3 Dotchart	515
14.5 Elemente einem bestehenden Diagramm hinzufügen	516
14.5.1 Koordinaten in einem Diagramm identifizieren	517
14.5.2 In beliebige Diagrammbereiche zeichnen	518
14.5.3 Punkte	519
14.5.4 Linien	520
14.5.5 Polygone	522
14.5.6 Funktionsgraphen	525
14.5.7 Text und mathematische Formeln	526
14.5.8 Achsen	528
14.5.9 Fehlerbalken	529
14.5.10 Rastergrafiken	532
14.6 Verteilungsdiagramme	534
14.6.1 Histogramm und Schätzung der Dichtefunktion	534
14.6.2 Stamm-Blatt-Diagramm	536
14.6.3 Boxplot	537
14.6.4 Stripchart	539
14.6.5 Quantil-Quantil-Diagramm	540
14.6.6 Empirische kumulierte Häufigkeitsverteilung	542
14.6.7 Kreisdiagramm	542
14.6.8 Gemeinsame Verteilung zweier Variablen	543
14.7 Daten interpolieren und fitten	547
14.7.1 Lineare Interpolation und LOESS-Glätter	547
14.7.2 Splines	548

Inhaltsverzeichnis

14.8 Multivariate Daten visualisieren	549
14.8.1 Höhenlinien und variable Datenpunktsymbole	550
14.8.2 Dreidimensionale Gitter und Streudiagramme	552
14.8.3 Bedingte Diagramme für mehrere Gruppen mit <code>ggplot2</code>	553
14.8.4 Bedingte Diagramme für mehrere Gruppen mit <code>lattice</code>	560
14.8.5 Matrix aus Streudiagrammen	561
14.8.6 Heatmap	563
14.9 Mehrere Diagramme in einem Grafik-Device darstellen	565
14.9.1 <code>layout()</code>	565
14.9.2 <code>par(mfrow, mfcoll, fig)</code>	567
14.9.3 <code>split.screen()</code>	569
15 R als Programmiersprache	571
15.1 Kontrollstrukturen	571
15.1.1 Fallunterscheidungen	571
15.1.2 Schleifen	574
15.2 Eigene Funktionen erstellen	577
15.2.1 Funktionskopf	577
15.2.2 Funktionsrumpf	578
15.2.3 Fehler behandeln	579
15.2.4 Rückgabewert und Funktionsende	581
15.2.5 Eigene Funktionen verwenden	581
15.2.6 Generische Funktionen	582
15.3 Funktionen analysieren und verbessern	584
15.3.1 Quelltext fremder Funktionen begutachten	584
15.3.2 Funktionen zur Laufzeit untersuchen	585
15.3.3 Effizienz von Auswertungen steigern	587
Literaturverzeichnis	590

Kapitel 1

Erste Schritte

1.1 Vorstellung

R ist eine freie und kostenlose Umgebung zur computergestützten statistischen Datenverarbeitung (Ihaka & Gentleman, 1996; R Development Core Team, 2014a): R integriert eine Vielzahl von Möglichkeiten, um Daten organisieren, transformieren, auswerten und visualisieren zu können. Dabei bezeichnet R sowohl das Programm selbst als auch die Sprache, in der die Auswertungsbefehle geschrieben werden.¹ In R bestehen Auswertungen nämlich aus einer Abfolge von Befehlen in Textform, die der Benutzer unter Einhaltung einer bestimmten Syntax selbst einzugeben hat. Jeder Befehl stellt dabei einen eigenen Auswertungsschritt dar, wobei eine vollständige Datenanalyse durch die Abfolge vieler solcher Schritte gekennzeichnet ist. So könnten Daten zunächst aus einer Datei gelesen und zwei Variablen zu einer neuen verrechnet werden, ehe eine Teilmenge von Beobachtungen ausgewählt und mit ihr ein statistischer Test durchgeführt wird, dessen Ergebnisse im Anschluss grafisch aufzubereiten sind.

1.1.1 Pro und Contra R

Während in Programmen, die über eine grafische Benutzeroberfläche gesteuert werden, die Auswahl von vorgegebenen Menüpunkten einen wesentlichen Teil der Arbeit ausmacht, ist es in R die aktive Produktion von Befehlsausdrücken. Diese Eigenschaft ist gleichzeitig ein wesentlicher Vorteil wie auch eine Herausforderung beim Einstieg in die Arbeit mit R. Folgende Aspekte lassen R für die Datenauswertung besonders geeignet erscheinen:

- Die befehlsgesteuerte Arbeitsweise erhöht durch die Wiederverwendbarkeit von Befehlssequenzen für häufig wiederkehrende Arbeitsschritte langfristig die Effizienz. Zudem steigt die Zuverlässigkeit von Analysen, wenn bewährte Bausteine immer wieder verwendet und damit Fehlerquellen ausgeschlossen werden.
- Die Möglichkeit zur Weitergabe von Befehlssequenzen an Dritte (gemeinsam mit empirischen Datensätzen) kann die Auswertung für andere transparent und überprüfbar machen. Auf diese Weise lassen sich Auswertungsergebnisse jederzeit exakt selbst und durch andere reproduzieren, wodurch die Auswertungsobjektivität steigt.

¹Genauer gesagt ist GNU R ursprünglich eine *open source* Implementierung der Sprache S (Becker, Chambers & Wilks, 1988). Der *open source* Programmcode zugrundeliegende Quelltext ist frei erhältlich, zudem darf die Software frei genutzt, verbreitet und verändert werden. Genaueres erläutert der Befehl `licence()`. Kommerzielle Varianten von R sind TIBCO TERR (TIBCO Software Inc., 2013) sowie Revolution R (Revolution Analytics, 2014).

- Als freies Open-Source-Programm wird R beständig von vielen Personen evaluiert, weiterentwickelt und verbessert (vgl. R Foundation for Statistical Computing, 2013 für eingesetzte Methoden der Qualitätssicherung). Da der Quelltext frei verfügbar ist und zudem viele Auswertungsfunktionen ihrerseits in R geschrieben sind, ist die Art der Berechnung statistischer Kennwerte vollständig transparent. Sie kann damit bei Interesse analysiert und auf Richtigkeit kontrolliert werden.
- Dank seines modularen Aufbaus bietet R die Möglichkeit, die Basisfunktionalität für spezielle Anwendungszwecke durch eigenständig entwickelte Zusatzkomponenten zu erweitern, von denen bereits mehrere tausend frei verfügbar sind.
- R-Auswertungen lassen sich in Dokumente einbetten, die beschreibenden Text, R-Befehle und die automatisch erzeugte Ausgabe dieser Befehle enthalten – inkl. der Diagramme (Gandrud, 2014; Xie, 2013). Die Texte können dabei sehr detailliert (in L^AT_EX-Syntax) oder mit einfachen Methoden (im aus Online-Wikis bekannten *markdown*-Format) formatiert werden. Aus demselben Ursprungstext lässt sich dann flexibel ebenso ein pdf- oder Word-Dokument wie auch eine HTML-Seite erzeugen (RStudio Inc, 2014b). Indem Hintergrundinformationen, Details zur Auswertung sowie die Ergebnisse und ihre Beschreibung kombiniert werden, bietet R eine umfassende Plattform für reproduzierbare statistische Analysen.
- Auch ohne tiefergehende Programmierkenntnisse lassen sich in R eigene Funktionen erstellen und so Auswertungen flexibel an individuelle Anforderungen anpassen. Da Funktionen in derselben Syntax wie normale Auswertungen geschrieben werden, sind dafür nur wenige Zusatzschritte zu erlernen, nicht aber eine eigene Makrosprache.

R hält für Anwender allerdings auch Hürden bereit, insbesondere für Einsteiger, die nur über Erfahrungen mit Programmen verfügen, die über eine grafische Benutzeroberfläche bedient werden:

- Die einzelnen Befehle und ihre Syntax zu erlernen, erfordert die Bereitschaft zur Einübung. Es muss zunächst ein Grundverständnis für die Arbeitsabläufe sowie ein gewisser „Wortschatz“ häufiger Funktionen und Konzepte geschaffen werden, ehe Daten umfassend ausgewertet werden können.
- Im Gegensatz zu Programmen mit grafischer Benutzeroberfläche müssen Befehle aktiv erinnert werden. Es stehen keine Gedächtnissstützen i. S. von Elementen einer grafischen Umgebung zur Verfügung, die interaktiv mögliche Vorgehensweisen anbieten und zu einem Wiedererkennen führen könnten. Dies betrifft sowohl die Auswahl von geeigneten Analysen wie auch die konkrete Anwendung bestimmter Verfahren.
- Im Gegensatz zur natürlichen Sprache ist die Befehlssteuerung nicht fehlertolerant, Befehle müssen also exakt richtig eingegeben werden. Dies kann zu Beginn der Beschäftigung mit R frustrierend und auch zeitraubend sein, weil schon vermeintlich unwesentliche Fehler verhindern, dass Befehle ausgeführt werden (vgl. Abschn. 1.2.8). Im Fall falsch eingegebener Befehle liefert R aber Fehlermeldungen, die Rückschlüsse auf die Ursache erlauben. Auch nimmt die Zahl der Syntaxfehler im Zuge der Beschäftigung mit R meist von allein deutlich ab.

- Bei der Analyse extrem großer Datenmengen besteht gegenwärtig das Problem, dass R im Gegensatz etwa zu S+ Datensätze zur Bearbeitung im Arbeitsspeicher vorhalten muss, was die Größe von praktisch auswertbaren Datensätzen einschränkt. Hinweise zu Lösungsansätzen sowie zur Ausnutzung paralleler Rechnerarchitekturen finden sich in Abschn. 15.3.3.

Startschwierigkeiten sind bei der Auseinandersetzung mit R also zu erwarten, sollten jedoch niemanden entmutigen: Auch wenn dies zu Beginn häufig Fehler nach sich zieht, ist vielmehr ein spielerisch-exploratives Kennenlernen wichtig, um ein besseres Verständnis der Grundprinzipien der Arbeit mit R zu entwickeln. Ein wichtiger Grundsatz ist dabei, sich zunächst inhaltlich zu überlegen, welche Teilschritte für eine konkrete Auswertung notwendig sind. Je konkreter die Teilschritte dieses gedanklichen Ablaufplans sind, desto einfacher lassen sich ihnen einzelne Befehlsbausteine zuordnen. Sind dann aus einfachen Auswertungen viele solcher Befehlsbausteine vertraut, lassen sie sich Schritt für Schritt zu komplexen Analysen zusammenstellen.

1.1.2 Typografische Konventionen

Zur besseren Lesbarkeit sollen zunächst einige typografische Konventionen für die Notation vereinbart werden. Um zwischen den Befehlen und Ausgaben von R sowie der zugehörigen Beschreibung innerhalb dieses Textes unterscheiden zu können, werden Befehle und Ausgaben im Schrifttyp **Schreibmaschine** dargestellt. Eine Befehlszeile mit zugehöriger Ausgabe könnte also z. B. so aussehen:

```
> 1 + 1  
[1] 2
```

Die erste Zeile bezeichnet dabei die Eingabe des Anwenders, die zweite Zeile die (in diesem Text bisweilen leicht umformatierte) Ausgabe von R. Fehlen Teile der Ausgabe im Text, ist dies mit ... als Auslassungszeichen angedeutet. Zeilenumbrüche innerhalb von R-Befehlen sind durch Pfeile in der Form **Befehl Anfang ↘ → Fortsetzung** gekennzeichnet, um deutlich zu machen, dass die getrennten Teile unmittelbar zusammen gehören. Platzhalter, wie z. B. **⟨Dateiname⟩**, die für einen Typ von Objekten stehen (hier Dateien) und mit einem beliebigen konkreten Objekt dieses Typs gefüllt werden können, werden in stumpfwinklige Klammern ⟨ ⟩ gesetzt. Schaltflächen des Programmfensters werden in serifloser Schrift dargestellt.

Internet-URLs, Tastenkombinationen sowie Dateien und Verzeichnisse werden im Schrifttyp **Schreibmaschine** dargestellt, wobei die Unterscheidung zu R-Befehlen aus dem Kontext hervorgeht.

1.1.3 R installieren

Zentrale Anlaufstelle für Nachrichten über die Entwicklung von R, für den download des Programms selbst, für Zusatzpakete sowie für frei verfügbare Literatur ist die R-Projektseite im WWW:

<http://www.r-project.org/>

Die folgenden Ausführungen beziehen sich auf die Installation des R-Basispaket unter Windows 8.1, vgl. dazu auch die offizielle Installationsanleitung (R Development Core Team, 2014c) und die Windows-FAQ (Ripley & Murdoch, 2014).² Für die Installationsdatei des Programms folgt man auf der Projektseite dem Verweis [Download, Packages / CRAN](#), der auf eine Übersicht von CRAN-servers verweist, von denen die Dateien erhältlich sind.³ Nach der Wahl eines CRAN-servers gelangt man über [Download and Install R / Download R for Windows: base](#) zum Verzeichnis mit der Installationsdatei `R-3.1.2-win.exe`, die auf dem eigenen Rechner zu speichern ist.⁴

Um R zu installieren, ist die gespeicherte Installationsdatei `R-<Version>-win.exe` auszuführen und den Aufforderungen des Setup-Assistenten zu folgen. Die Installation setzt voraus, dass der Benutzer ausreichende Schreibrechte auf dem Computer besitzt, weshalb es u. U. notwendig ist, R als Administrator zu installieren. Wenn keine Änderungen am Installationsordner von R vorgenommen wurden, sollte R daraufhin im Verzeichnis `C:\Programme\R\R-<Version>\` installiert und eine zugehörige Verknüpfung auf dem desktop sowie im Startmenü vorhanden sein.

Unter MacOS verläuft die Installation analog, wobei den entsprechenden CRAN-links zur Installationsdatei zu folgen ist. R ist in allen gängigen Linux-Distributionen enthalten und kann dort am einfachsten direkt über den jeweils verwendeten Paketmanager installiert werden.

1.1.4 Grafische Benutzeroberflächen

Obwohl nicht zwingend notwendig, empfiehlt es sich, eine grafische Umgebung zur Befehlseingabe zu installieren, die komfortabler als die in R mitgelieferte Oberfläche ist. Eine alternative grafische Umgebung setzt dabei voraus, dass R selbst bereits installiert wurde.

- Die auf R zugeschnittene Entwicklungsumgebung RStudio (RStudio Inc, 2014c) ist frei verfügbar, einfach zu installieren, läuft unter mehreren Betriebssystemen mit einer konsistenten Oberfläche und erleichtert häufige Arbeitsschritte (vgl. Abschn. 1.2.1). Zudem unterstützt RStudio besonders gut die Möglichkeiten, Dokumente mit eingebetteten R-Auswertungen zu erstellen (Gandrud, 2014; RStudio Inc, 2014b).
- Ähnlich wie RStudio liefert die plattformunabhängige und freie Entwicklungsumgebung Architect (OpenAnalytics BVBA, 2014) eine grafische Oberfläche zur Arbeit mit R, die weit komfortabler als die R-eigene und mehr Funktionen bietet. Architect bündelt die Entwicklungsumgebung Eclipse mit dem StatET Plugin (Wahlbrink, 2014) und ist deutlich einfacher zu installieren als beide Programme separat.

²Abgesehen von der Oberfläche und abweichenden Pfadangaben bestehen nur unwesentliche Unterschiede zwischen der Arbeit mit R unter verschiedenen Betriebssystemen.

³CRAN steht für *Comprehensive R Archive Network* und bezeichnet ein Netzwerk von mehreren *mirror servers* mit gleichem Angebot, die die aktuellen Dateien und Informationen zu R anbieten. Aus der Liste der verfügbaren server sollte einer nach dem Kriterium der geografischen Nähe ausgewählt werden.

⁴`R-3.1.2-win.exe` ist die im November 2014 aktuelle Version von R für Windows. 3.1.2 ist die Versionsnummer. Bei neueren Versionen sind leichte, für den Benutzer jedoch üblicherweise nicht merkliche Abweichungen zur in diesem Manuskript beschriebenen Arbeitsweise von Funktionen möglich.

- Der freie und plattformunabhängige Texteditor GNU Emacs lässt sich ebenso wie XEmacs mit dem Zusatzpaket ESS (*Emacs Speaks Statistics*, Rossini et al., 2014) zu einer textbasierten Entwicklungsumgebung für R erweitern. Zielgruppe sind vor allem fortgeschrittene Anwender, die bereits den sehr mächtigen, aber auch etwas eigenen Editor Emacs kennen.
- Unter Windows eignet sich etwa Tinn-R (Faria, 2014), um lediglich den R-eigenen Texteditor zu ersetzen.
- Das Zusatzpaket JGR (Helbig, Urbanek & Fellows, 2013, vgl. Abschn. 1.2.7) stellt eine plattformunabhängige Oberfläche bereit, die der R-eigenen Oberfläche unter Windows sehr ähnlich ist.
- Schließlich existieren Programme, die Analysefunktionen über grafische Menüs und damit ohne Befehlseingabe nutzbar machen, darunter Rcmdr (Fox, 2005) und Deducer (Fellows, 2014). Diese Programme bieten derzeit jedoch nur Zugang zu einem Teil der Möglichkeiten, die R bereitstellt.

1.1.5 Weiterführende Informationsquellen und Literatur

Häufige Fragen zu R sowie speziell zur Verwendung von R unter Windows werden in den FAQs (frequently asked questions) beantwortet (Hornik, 2014; Ripley & Murdoch, 2014). Für individuelle Fragen existiert die Mailing-Liste R-help, deren Adresse auf der Projektseite unter [R Project / Mailing Lists](#) genannt wird. Bevor sie für eigene Hilfegesuche genutzt wird, sollte aber eine umfangreiche selbständige Recherche vorausgehen. Zudem sind die Hinweise des posting-guides zu beherzigen. Beides gilt auch für das hilfreiche Web-Forum StackOverflow:

<http://www.r-project.org/posting-guide.html>
<http://stackoverflow.com/tags/R>

Die Suche innerhalb von Beiträgen auf der Mailing-Liste, sowie innerhalb von Funktionen und der Hilfe erleichtert die folgende auf R-Inhalte spezialisierte Seite:

<http://www.rdocumentation.org/>

Unter dem link [Documentation / Manuals](#) auf der Projektseite von R findet sich die vom R Entwickler-Team herausgegebene offizielle Dokumentation. Sie liefert einerseits einen umfassenden, aber sehr konzisen Überblick über R selbst (Venables, Smith & the R Development Core Team, 2014) und befasst sich andererseits mit Spezialthemen wie dem Datenaustausch mit anderen Programmen (vgl. Abschn. 4.2). Weitere, von Anwendern beigesteuerte Literatur zu R findet sich auf der Projektseite unter dem Verweis [Documentation / Other](#). Darüber hinaus existiert eine Reihe von Büchern mit unterschiedlicher inhaltlicher Ausrichtung, die R-Projektseite bietet dazu unter [Documentation / Books](#) eine laufend aktualisierte Literaturübersicht:

- Empfehlenswerte Einführungen in R anhand grundlegender statistischer Themen findet man etwa bei Dalgaard (2008); Maindonald und Braun (2010) sowie Verzani (2014). Everitt und Hothorn (2010) sowie Venables und Ripley (2002) behandeln den Einsatz von R für fortgeschrittene statistische Anwendungen.

- Für die Umsetzung spezieller statistischer Themen existieren eigene Bücher – etwa zu Regressionsmodellen (Fox & Weisberg, 2011; Harrell Jr, 2015), multivariaten Verfahren (Everitt & Hothorn, 2011), gemischten Modellen (Gałecki & Burzykowski, 2013; Long, 2012; Pinheiro & Bates, 2000), Resampling-Verfahren (Chernik & LaBudde, 2011; Chihara & Hesterberg, 2011) und der Analyse von Zeitreihen (Shumway & Stoffer, 2011).
- Muenchen (2011) sowie Muenchen und Hilbe (2010) richten sich speziell an Umsteiger von anderen Statistikpaketen wie SPSS, SAS oder Stata und vertiefen die Zusammenarbeit von R mit diesen Programmen.
- Das Gebiet der Programmierung mit R wird von Chambers (2008); Wickham (2014a) und Ligges (2014) thematisiert.
- Xie (2013) sowie Gandrud (2014) behandeln, wie mit R dynamische Dokumente erstellt werden können, die Berechnungen, Ergebnisse und Beschreibungen integrieren.

1.2 Grundlegende Elemente

1.2.1 R Starten, beenden und die Konsole verwenden

Nach der Installation lässt sich R unter Windows über die bei der Installation erstellte Verknüpfung auf dem desktop bzw. im Startmenü starten. Hierdurch öffnen sich zwei Fenster: ein großes, das Programmfenster, und darin ein kleineres, die *Konsole*. Unter MacOS und Linux ist die Konsole das einzige sich öffnende Fenster.

Der Arbeitsbereich der empfehlenswerten Entwicklungsumgebung RStudio gliedert sich in vier Regionen (Abb. 1.1):⁵

- Links unten befindet sich die Konsole zur interaktiven Arbeit mit R. Befehle können direkt auf der Konsole eingegeben werden, außerdem erscheint die Ausgabe von R hier.
- Links oben öffnet sich ein Editor, in dem ein eigenes Befehlsskript erstellt oder ein bereits vorhandenes Befehlsskript geöffnet werden kann. Ein Skript ist dabei eine einfache Textdatei mit einer Sammlung von nacheinander abzuarbeitenden Befehlen (vgl. Abschn. 4.1). Die Befehle eines Skripts können im Editor markiert und mit dem icon Run bzw. mit der Tastenkombination **Strg+r** an die Konsole gesendet werden.
- Rechts oben befinden sich zwei Tabs: **Workspace** zeigt alle derzeit verfügbaren Objekte an – etwa Datensätze oder einzelne Variablen, die sich für Auswertungen nutzen lassen (vgl. Abschn. 1.2.3). **History** speichert als Protokoll eine Liste der schon aufgerufenen Befehle.
- Rechts unten erlaubt es **Files**, die Ordner der Festplatte anzuzeigen und zu navigieren. Im **Plots**-Tab öffnen sich die erstellten Diagramme, **Packages** informiert über die verfügbaren Zusatzpakete (vgl. Abschn. 1.2.7), und **Help** erlaubt den Zugriff auf das Hilfesystem (vgl. Abschn. 1.2.6).

⁵RStudio wird derzeit mit hohem Tempo weiterentwickelt. Es ist deshalb möglich, dass im Laufe der Zeit Aussehen und Funktionalität in Details von der folgenden Beschreibung abweichen.

Kapitel 1 Erste Schritte

- RStudio lässt sich über den Menüeintrag Tools: Global Options stark an die eigenen Präferenzen anpassen.

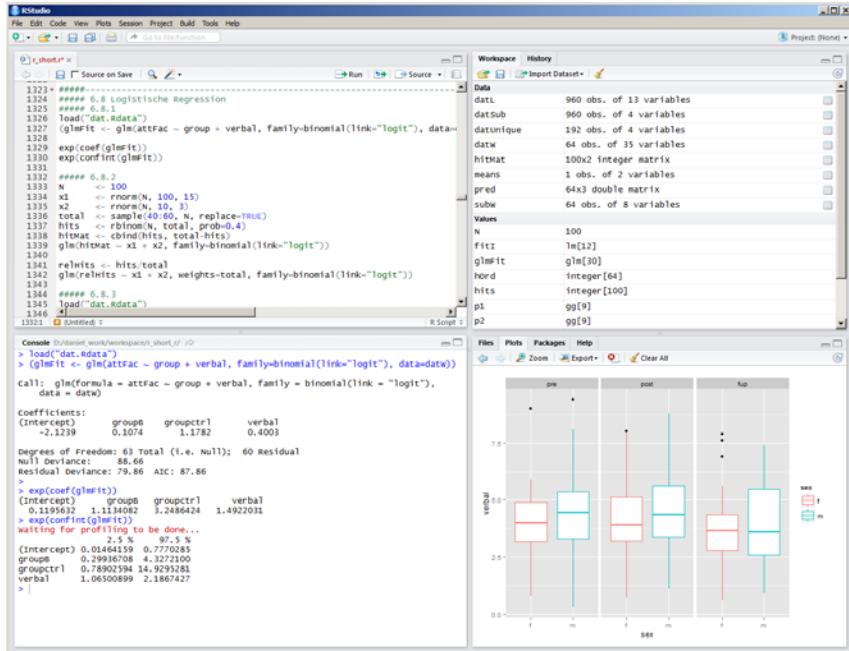


Abbildung 1.1: Oberfläche der Entwicklungsumgebung RStudio

Auf der Konsole werden im interaktiven Wechsel von eingegebenen Befehlen und der Ausgabe von R die Verarbeitungsschritte vorgenommen.⁶ Hier erscheint nach dem Start unter einigen Hinweisen zur installierten R-Version hinter dem *Prompt*-Zeichen > ein Cursor.⁷ Der Cursor signalisiert, dass Befehle vom Benutzer entgegengenommen und verarbeitet werden können. Das Ergebnis einer Berechnung wird in der auf das Prompt-Zeichen folgenden Zeile ausgegeben, nachdem ein Befehl mit der **Return** Taste beendet wurde. Dabei wird in eckigen Klammern oft zunächst die laufende Nummer des ersten in der jeweiligen Zeile angezeigten Objekts aufgeführt. Anschließend erscheint erneut ein Prompt-Zeichen, hinter dem neue Befehle eingegeben werden können.

```
> 1 + 1
[1] 2
```

```
>
```

⁶Für automatisierte Auswertungen vgl. Abschn. 4.1. Die Ausgabe lässt sich mit der `sink()` Funktion entweder gänzlich oder i. S. eines Protokolls aller Vorgänge als Kopie in eine Datei umleiten (Argument `split=TRUE`). Befehle des Betriebssystems sind mit `shell("Befehl")` ausführbar, so können etwa die Netzwerkverbindungen mit `shell("netstat")` angezeigt werden.

⁷Weitere Details zur installierten R-Version sowie zur Systemumgebung liefern die Funktionen `Sys.info()`, `sessionInfo()`, `R.Version()` sowie der Befehl `.Platform`.

Pro Zeile wird im Normalfall ein Befehl eingegeben. Sollen mehrere Befehle in eine Zeile geschrieben werden, so sind sie durch ein Semikolon ; zu trennen. Das Symbol # markiert den Beginn eines *Kommentars* und verhindert, dass der dahinter auftauchende Text in derselben Zeile als Befehl interpretiert wird.

War die Eingabe nicht korrekt, erscheint eine Fehlermeldung mit einer Beschreibung der Ursache. Abschnitt 1.2.8 erläutert typische Fehlerquellen. Befehle können auch Warnungen verursachen, die die Auswertung zwar nicht wie Fehler verhindern, aber immer daraufhin untersucht werden sollten, ob sie ein Symptom für falsche Berechnungen sind.

Das folgende Beispiel soll eine kleine Auswertung in Form mehrerer aufeinanderfolgender Arbeitsschritte demonstrieren. An dieser Stelle ist es dabei nicht wichtig, schon zu verstehen, wie die einzelnen Befehle funktionieren. Vielmehr soll das Beispiel zeigen, wie eine realistische Sequenz von Auswertungsschritten inkl. der von R erzeugten Ausgabe aussehen kann. Die Analyse bezieht sich auf den Datensatz **Duncan** aus dem Paket **car** (Fox & Weisberg, 2014), das zunächst zu installieren ist (vgl. Abschn. 1.2.7). Der Datensatz speichert Daten von Personen verschiedener Berufe. Ein Beruf kann dabei zur Gruppe **bc** (blue collar), **wc** (white collar) oder **prof** (professional) gehören – die Stufen der Variable **type**. Erhoben wurde der prozentuale Anteil von Personen eines Berufs mit einem hohen Einkommen (Variable **income**), einem hohen Ausbildungsgrad (**education**) und einem hohen Prestige (**prestige**). Jede Zeile enthält die Daten jeweils eines Berufs, die Daten jeweils einer Variable finden sich in den Spalten des Datensatzes.

```
> data(Duncan, package="car")           # lade Datensatz Duncan aus Paket car
> head(Duncan)                      # gib die ersten Zeilen von Duncan aus
   type income education prestige
accountant  prof      62        86     82
pilot        prof      72        76     83
architect    prof      75        92     90
author       prof      55        90     76
chemist      prof      64        86     90
minister     prof      21        84     87
```

Zunächst sollen wichtige deskriptive Kennwerte von **income** in nach Gruppen getrennten boxplots dargestellt und dabei auch die Rohdaten selbst abgebildet werden (Abb. 1.2). Es folgt die Berechnung der Gruppenmittelwerte für **education** und die Korrelationsmatrix der drei erhobenen Variablen. Als inferenzstatistische Auswertung schließt sich eine Varianzanalyse mit der Variable **prestige** und dem Gruppierungsfaktor **type** an. Im folgenden *t*-Test der Variable **education** sollen nur die Gruppen **bc** und **wc** berücksichtigt werden.

```
# nach Gruppen getrennte boxplots und Rohdaten der Variable income
> boxplot(income ~ type, data=Duncan,
+           main="Anteil hoher Einkommen pro Berufsgruppe")

> stripchart(income ~ type, data=Duncan, pch=20, vert=TRUE, add=TRUE)

#####
# Gruppenmittelwerte von education
> aggregate(education ~ type, FUN=mean, data=Duncan)
```

```

type   education
1    bc    25.33333
2  prof   81.33333
3    wc   61.50000

#####
# Korrelationsmatrix der Variablen income, education, prestige
> with(Duncan, cor(cbind(income, education, prestige)))
      income   education   prestige
income     1.0000000  0.7245124  0.8378014
education  0.7245124  1.0000000  0.8519156
prestige   0.8378014  0.8519156  1.0000000

#####
# einfaktorielle Varianzanalyse von prestige in Gruppen type
> anova(lm(prestige ~ type, data=Duncan))
Analysis of Variance Table
Response: prestige
            Df  Sum Sq Mean Sq F value    Pr(>F)
type          2  33090   16545   65.571 1.207e-13 ***
Residuals   42  10598      252
              

#####
# nur Berufe der Gruppen bc oder wc auswählen
BCandWC <- with(Duncan, (type == "bc") | (type == "wc"))

#####
# linksseitiger t-Test der Variable education
# für die zwei ausgewählten Berufsgruppen bc und wc
t.test(education ~ type, alternative="less", data=Duncan, subset=BCandWC)
Welch Two Sample t-test
data: education by type
t = -4.564, df = 5.586, p-value = 0.002293
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
 -Inf -20.56169

sample estimates:
mean in group bc  mean in group wc
      25.33333       61.50000

```

Einzelne Befehlsbestandteile müssen meist nicht unbedingt durch Leerzeichen getrennt werden, allerdings erhöht dies die Übersichtlichkeit und ist mitunter auch erforderlich. Insbesondere das Zuweisungssymbol `<-` sollte stets von Leerzeichen umschlossen sein. Wenn ein Befehl die sichtbare Zeilenlänge der Konsole überschreitet, verlängert R die Zeile automatisch, schränkt aber auf diese Weise die Sichtbarkeit des Befehlsbeginns ein. Ein langer Befehl kann darum auch durch Drücken der `Return` Taste in der nächsten Zeile fortgesetzt werden, solange er

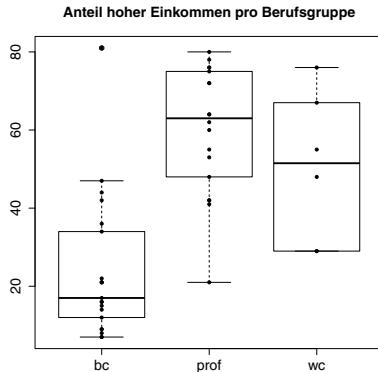


Abbildung 1.2: Daten des Datensatzes *Duncan*: Anteil der Angehörigen eines Berufs mit hohem Einkommen in Abhängigkeit vom Typ des Berufs

syntaktisch nicht vollständig ist – z. B. wenn eine geöffnete Klammer noch nicht geschlossen wurde. Es erscheint dann ein + zu Beginn der folgenden Zeile, um zu signalisieren, dass sie zur vorangehenden gehört. Falls dieses + ungewollt in der Konsole auftaucht, muss der Befehl entweder vervollständigt oder mit der ESC Taste (Windows) bzw. der Strg+c Tastenkombination (Linux) abgebrochen werden.

```
> 2 * (4
+ -5)
[1] -2
```

In der Konsole können Befehle mit der Maus markiert und (unter Windows) über die Tastenkombination Strg+c in die Zwischenablage hinein bzw. mit Strg+v aus der Zwischenablage heraus in die Befehlszeile hinein kopiert werden.

Eine weitere Funktionalität der Konsole ist die Tab Vervollständigung. Wird der Anfang eines Befehlsnamens in die Konsole eingegeben und dann die Tabulator-Taste gedrückt, vervollständigt R den begonnenen Befehl. Gibt es dafür mehrere Kandidaten, ist zweimal Tab zu drücken, um eine Auflistung möglicher Ergänzungen zu erhalten. Es existieren noch weitere vergleichbare Kurzbefehle, über die das RStudio Menü Help: Keyboard Shortcuts informiert.

R wird entweder über den Befehl q() in der Konsole oder durch Schließen des Programmfensters beendet. Zuvor erscheint die Frage, ob man den *workspace*, also alle erstellten Daten, im aktuellen Arbeitsverzeichnis speichern möchte (vgl. Abschn. 1.2.2, 1.2.3).

1.2.2 Einstellungen

R wird immer in einem *Arbeitsverzeichnis* ausgeführt, das sich durch getwd() ausgeben lässt. In der Voreinstellung handelt es sich um das Heimverzeichnis des Benutzers. Alle während einer Sitzung gespeicherten Dateien werden, sofern nicht explizit anders angegeben, in diesem Verzeichnis abgelegt. Wenn Informationen aus Dateien geladen werden sollen, greift R ebenfalls

auf dieses Verzeichnis zu, sofern kein anderes ausdrücklich genannt wird. Um das voreingestellte Arbeitsverzeichnis zu ändern, existieren folgende Möglichkeiten:

- Unter Windows durch einen Rechtsklick auf die auf dem desktop erstellte Verknüpfung mit R: Eigenschaften auswählen und anschließend bei Ausführen in ein neues Arbeitsverzeichnis angeben. Dies führt dazu, dass R von nun an immer im angegebenen Arbeitsverzeichnis startet, wenn die Verknüpfung zum Programmstart verwendet wird.
- Im Menü der R-eigenen Oberfläche unter Windows kann Datei: Verzeichnis wechseln gewählt und anschließend ein neues Arbeitsverzeichnis angegeben werden. Dieser Schritt ist dann nach jedem Start von R zu wiederholen. Eine analoge Funktion findet sich in RStudio im Menüpunkt Session: Set Working Directory.
- In der Konsole lässt sich das aktuelle Arbeitsverzeichnis mit `setwd("<Pfad>")` ändern, unter Windows also z.B. mit `setwd("c:/work/r/")` (vgl. Kap. 4.3 für die möglichen Formen von Pfadangaben).

R wird mit einer Reihe von Voreinstellungen gestartet, die sich über selbst editierbare Textdateien steuern lassen, über die `?Startup` Auskunft gibt (R Development Core Team, 2014c). Sollen etwa bestimmte Pakete in jeder Sitzung geladen werden (vgl. Abschn. 1.2.7), können die entsprechenden `library()` Befehle in die Datei `Rprofile.site` im `etc/` Ordner des Programmordners geschrieben werden.⁸ Gleiches gilt für befehlsübergreifende Voreinstellungen, die mit `getOption()` angezeigt und mit `options()` verändert werden können.

```
>getOption("<Option>")          # gibt aktuellen Wert für <Option> aus  
>options(<Option>=<Wert>)      # setzt <Option> auf <Wert>
```

Mit `options(width=<Anzahl>)` kann z. B. festgelegt werden, mit wie vielen Zeichen pro Zeile die Ausgabe von R erfolgt. `options()` liefert dabei auf der Konsole unsichtbar den Wert zurück, den die Einstellung vor ihrer Änderung hatte. Wird dieser Wert in einem Objekt gespeichert, kann die Einstellung später wieder auf ihren ursprünglichen Wert zurückgesetzt werden.

```
>getOption("width")           # aktuellen Wert für Option anzeigen  
[1] 116  
  
> op <- options(width=70)    # Option ändern und alten Wert in op speichern  
> options(op)              # Option auf alten Wert zurücksetzen
```

1.2.3 Umgang mit dem workspace

R speichert automatisch alle während einer Sitzung ausgeführten Befehle und neu erstellten Daten temporär zwischen. Die Daten liegen dabei in einem *workspace*, der auch als *Umgebung* (environment) bezeichnet wird. So können Kopien von Befehlen und Daten nachträglich in externen Dateien abgespeichert und bei einer neuen Sitzung wieder genutzt werden. Auf bereits

⁸Zudem kann jeder Benutzer eines Computers die Datei `.Rprofile` in seinem Heimverzeichnis anlegen – unter Windows im Ordner **Eigene Dokumente**. In dieser Datei können auch die Funktionen namens `.First` bzw. `.Last` mit beliebigen Befehlen definiert werden, die dann beim Start als erstes bzw. beim Beenden als letztes ausgeführt werden (vgl. Abschn. 15.2).

eingegebene Befehle kann auch bereits während derselben Sitzung erneut zugegriffen werden: Über die Pfeiltasten nach oben und unten lassen sich verwendete Befehle wieder aufrufen. Diese Funktion wird im folgenden als Befehlshistorie bezeichnet. Eine Übersicht der letzten Befehle liefert die Funktion `history()`, in RStudio finden sich die Befehle im Tab History.

Der erwähnte workspace trägt den Namen `.GlobalEnv`. Neben ihm existieren noch weitere Umgebungen, die abgekapselt voneinander ebenfalls Objekte speichern. Da ein Nutzer auf die Objekte aller Umgebungen zugreifen kann, tritt dieser Umstand nur selten explizit zutage, ist jedoch manchmal wichtig: Objekte, die in unterschiedlichen Umgebungen liegen, können denselben Namen tragen, ohne dass ihre Inhalte wechselseitig überschrieben würden (vgl. Abschn. 1.3.1). Die Umgebungen sind intern linear geordnet. Die Reihenfolge, in der R die Umgebungen nach Objekten durchsucht, ist der *Suchpfad*, der von `search()` ausgegeben wird. Existieren mehrere Objekte desselben Namens in unterschiedlichen Umgebungen, bezeichnet der einfache Name das Objekt in der früheren Umgebung. Welche Objekte eine bestimmte Umgebung speichert, lässt sich auf der Konsole mit `ls()` feststellen, während RStudio dafür den Environment Tab besitzt.

```
> ls(name = "(Umgebung)", pattern = "(Suchmuster)")
```

In der Voreinstellung `ls()` werden die Objekte der derzeit aktiven Umgebung aufgelistet, bei einem Aufruf von der Konsole i. d. R. also jene aus `.GlobalEnv`, der ersten Umgebung im Suchpfad. Alternativ kann für `name` der Name einer Umgebung oder aber ihre Position im Suchpfad angegeben werden. Über `pattern` lässt sich ein Muster für den Objektnamen spezifizieren, so dass nur solche Objekte angezeigt werden, für die das Muster passt. Im einfachsten Fall könnte dies ein Buchstabe sein, den der Objektname enthalten muss, kompliziertere Muster sind über reguläre Ausdrücke möglich (vgl. Abschn. 2.12.4).

```
> ls()                                # nenne alle Objekte der aktuellen Umgebung
[1] "BCandWC" "Duncan"

# Objekte der Standard-Umgebung, die ein großes C im Namen tragen
> ls(".GlobalEnv", pattern = "C")
[1] "BCandWC"
```

Es gibt mehrere, in ihren Konsequenzen unterschiedliche Wege, Befehle und Daten der aktuellen Sitzung zu sichern. Eine Kopie des aktuellen workspace sowie der Befehlshistorie wird etwa gespeichert, indem man die beim Beenden von R erscheinende diesbezügliche Frage bejaht. Als Folge wird – falls vorhanden – der sich in diesem Verzeichnis befindende, bereits während einer früheren Sitzung gespeicherte workspace automatisch überschrieben. R legt bei diesem Vorgehen zwei Dateien an: eine, die lediglich die erzeugten Daten der Sitzung enthält (Datei `.RData`) und eine mit der Befehlshistorie (Datei `.Rhistory`). Der so gespeicherte workspace wird beim nächsten Start von R automatisch geladen.

Der workspace kann in RStudio im Environment Tab über das Disketten-icon unter einem frei wählbaren Namen gespeichert werden, um früher angelegte Dateien nicht zu überschreiben. Über das Ordner-icon im Environment Tab lässt sich eine Workspace-Datei wieder laden. Dabei ist zu beachten, dass zuvor definierte Objekte von jenen Objekten aus dem geladenen workspace überschrieben werden, die denselben Namen tragen. Die manuell gespeicherten workspaces werden bei einem Neustart von R nicht automatisch geladen.

Um die Befehlshistorie unter einem bestimmten Dateinamen abzuspeichern, wählt man den History Tab, der ebenfalls ein Disketten-icon besitzt. In der Konsole stehen zum Speichern und Laden der Befehlshistorie `savehistory("<Dateiname>")` und `loadhistory("<Dateiname>")` als Funktionen zur Verfügung.

1.2.4 Einfache Arithmetik

In R sind die grundlegenden arithmetischen Operatoren, Funktionen und Konstanten implementiert, über die auch ein Taschenrechner verfügt. Für eine Übersicht vgl. die mit dem Befehl `?Syntax` aufzurufende Hilfe-Seite. Punktrechnung geht dabei vor Strichrechnung, das Dezimaltrennzeichen ist unabhängig von den Ländereinstellungen des Betriebssystems immer der Punkt. Nicht ganzzahlige Werte werden in der Voreinstellung mit sieben relevanten Stellen ausgegeben, was mit dem Befehl `options(digits=<Anzahl>)` veränderbar ist.

Alternativ können Zahlen auch in wissenschaftlicher, also verkürzter Exponentialschreibweise ausgegeben werden.⁹ Dabei ist z. B. der Wert $2e-03$ als $2 \cdot 10^{-3}$, also $\frac{2}{10^3}$, mithin 0.002 zu lesen. Auch die Eingabe von Zahlen ist in diesem Format möglich.

Tabelle 1.1: Arithmetische Funktionen, Operatoren und Konstanten

Operator / Funktion / Konstante	Bedeutung
<code>+</code> , <code>-</code>	Addition, Subtraktion
<code>*</code> , <code>/</code>	Multiplikation, Division
<code>%/%</code>	ganzzahlige Division (ganzzahliges Ergebnis einer Division ohne Rest)
<code>%%</code>	Modulo Division (Rest einer ganzzahligen Division, verallgemeinert auf Dezimalzahlen ¹⁰)
<code>~</code>	potenzieren
<code>sign()</code>	Vorzeichen (-1 bei negativen, 1 bei positiven Zahlen, 0 bei der Zahl 0)
<code>abs()</code>	Betrag
<code>sqrt()</code>	Quadratwurzel
<code>round(<Zahl>, digits=<Anzahl>)</code>	runden (mit Argument zur Anzahl der Dezimalstellen) ¹¹
<code>floor()</code> , <code>ceiling()</code> , <code>trunc()</code>	auf nächsten ganzzahligen Wert abrunden, aufrunden, tranchieren (Nachkommastellen abschneiden)

⁹Sofern diese Formatierung nicht mit `options(scipen=999)` ganz unterbunden wird. Allgemein kann dabei mit ganzzahlig positiven Werten für `scipen` (*scientific penalty*) die Schwelle erhöht werden, ab der R die wissenschaftliche Notation für Zahlen verwendet (vgl. `?options`).

¹⁰Der Dezimalteil einer Dezimalzahl ergibt sich also als `<Zahl> %% 1`.

¹¹R rundet in der Voreinstellung nicht nach dem vielleicht vertrauten Prinzip des kaufmännischen Rundens, sondern *unverzerrt* (Bronstein, Semendjajew, Musiol & Mühlig, 2012). Durch negative Werte für `digits` kann auch auf Zehnerpotenzen gerundet werden. `signif()` rundet auf eine bestimmte Anzahl signifikanter Stellen.

Tabelle 1.1: (Forts.)

<code>log()</code> , <code>log10()</code> , <code>log2()</code> , <code>log(<Zahl>, base=<Basis>)</code>	natürlicher Logarithmus, Logarithmus zur Basis 10, zur Basis 2, zu beliebiger Basis
<code>exp()</code>	Exponentialfunktion
<code>exp(1)</code>	Eulersche Zahl e
<code>sin()</code> , <code>cos()</code> , <code>tan()</code> , <code>asin()</code> , <code>acos()</code> , <code>atan()</code> , <code>atan2()</code>	trigonometrische Funktionen sowie ihre Umkehrfunktionen (Argument im Bogenmaß)
<code>factorial()</code>	Fakultät
<code>pi</code>	Kreiszahl π
<code>Inf</code> , <code>-Inf</code>	∞ , $-\infty$ (infinity)
<code>NA</code>	fehlender Wert (not available)
<code>NaN</code>	nicht definiert (not a number), verhält sich weitestgehend wie NA
<code>NULL</code>	leere Menge

Die in Tab. 1.1 aufgeführten Funktionen, Operatoren und Konstanten sind frei kombinierbar und können auch ineinander verschachtelt werden. Um die Reihenfolge der Auswertung eindeutig zu halten, sollten in Zweifelsfällen Klammern gesetzt werden.¹²

```
> 12^2 + 1.5*10
[1] 159

> sin(pi/2) + sqrt(abs(-4))
[1] 3
```

`<Realteil>+<Imaginärteil>i` ist die Notation zur Eingabe komplexer Zahlen. `0+1i` ist also die imaginäre Zahl i , `-1+0i` hingegen die reelle Zahl -1 in komplexer Schreibweise. Den Realteil einer komplexen Zahl liefert `Re(<Zahl>)`, den Imaginärteil `Im(<Zahl>)`. Die Funktionen `Mod(<Zahl>)` und `Arg(<Zahl>)` geben die Polarkoordinaten in der komplexen Ebene aus, die komplexe Konjugierte ermittelt `Conj(<Zahl>)`. In der Voreinstellung rechnet R mit reellen Zahlen, aus diesem Grund erzeugt etwa `sqrt(-1)` die Ausgabe `NaN`, während `sqrt(-1+0i)` mit `0+1i` das richtige Ergebnis i liefert.

```
> exp(1)^((0+1i) * pi)          # komplexe Zahl (Eulersche Identität)
[1] -1+0i
```

Um zu überprüfen, ob ein Objekt einen „besonderen“ numerischen Wert speichert, stellt R Funktionen bereit, die nach dem Muster `is.(Prüfwert)(<Zahl>)` aufgebaut sind und einen Wahrheitswert zurückgeben (vgl. Abschn. 1.3.6).

```
> is.infinite(<Zahl>)           # ist <Zahl> unendlich?
> is.finite(<Zahl>)            # ist <Zahl> gültige endliche Zahl?
> is.nan(<Zahl>)              # ist <Zahl> nicht definiert?
> is.na(<Zahl>)                # ist <Zahl> ein fehlender Wert?
```

¹²Für die zur Bestimmung der Ausführungsreihenfolge wichtige Assoziativität von Operatoren vgl. `?Syntax`.

```
> is.null(<Zahl>)                      # ist <Zahl> die leere Menge?

> is.infinite(1/0)
[1] TRUE

> is.nan(0/0)
[1] TRUE
```

Die Funktion `is.finite(<Objekt>)` liefert nur für solche Objekte TRUE zurück, die eine gültige Zahl sind, dagegen ergeben neben `-Inf` und `Inf` auch Zeichenketten, NA, NaN und NULL das Ergebnis FALSE.

1.2.5 Funktionen mit Argumenten aufrufen

Beim Aufruf von Funktionen in R sind die Werte, die der Funktion als Berechnungsgrundlage dienen, in runde Klammern einzuschließen: `<Funktionsname>(<Argumentliste>)`. Die Argumentliste besteht aus Zuweisungen an Argumente in der Form `<Argumentname>=<Wert>`, die der Funktion die notwendigen Eingangsinformationen liefern. Es können je nach Funktion ein oder mehrere durch Komma getrennte Argumente angegeben werden, die ihrerseits obligatorisch oder nur optional sein können.¹³ Auch wenn eine Funktion keine Argumente besitzt, müssen die runden Klammern vorhanden sein, z. B. `q()`.¹⁴

Argumente sind benannt und machen so ihre Bedeutung für die Arbeitsweise der Funktion deutlich. Um z. B. eine Zahl zu runden, muss der Funktion `round(<Zahlen>, digits=0)` mindestens ein zu rundender Wert übergeben werden, z. B. `round(1.27)`. Weiterhin besteht die Möglichkeit, über das zusätzliche Argument `digits` die gewünschte Anzahl an Nachkommastellen zu bestimmen: Mit `round(pi, digits=2)` wird die Zahl π auf zwei Dezimalstellen gerundet. Das Argument `digits` ist optional, wird es nicht angegeben, kommt der auf 0 voreingestellte Wert (*default*) zur Rundung auf ganze Zahlen zum tragen.

Der Name von Argumenten muss nicht unbedingt vollständig angegeben werden, wenn eine Funktion aufgerufen wird – eine Abkürzung auf den zur eindeutigen Identifizierung notwendigen Namensanfang reicht aus.¹⁵ Von dieser Möglichkeit sollte jedoch mit Blick auf die Verständlichkeit des Funktionsaufrufs kein Gebrauch gemacht werden. Fehlt der Name eines Arguments ganz, so erfolgt die Zuordnung eines im Funktionsaufruf angegebenen Wertes über seine Position in der Argumentliste: Beim Befehl `round(pi, 3)` wird die 3 als Wert für das `digits` Argument interpretiert, weil sie an zweiter Stelle steht. Allgemein empfiehlt es sich, nur den Namen des ersten Hauptarguments wegzulassen und die übrigen Argumentnamen aufzuführen, insbesondere, wenn viele Argumente an die Funktion übergeben werden können.

¹³In diesem Text werden nur die wichtigsten Argumente der behandelten Funktionen vorgestellt, eine vollständige Übersicht liefert jeweils `args(<Funktionsname>)` sowie die zugehörige Hilfe-Seite `?<Funktionsname>`.

¹⁴In R sind Operatoren wie `+`, `-`, `*` oder `/` Funktionen, für die lediglich eine bequemere und vertrautere Kurzschrifweisreise zur Verfügung steht. Operatoren lassen sich auch in der Präfix-Form benutzen, wenn sie in Anführungszeichen gesetzt werden. So ist `"/"(1, 10)` äquivalent zu `1/10`.

¹⁵Gleiches gilt für die Werte von Argumenten, sofern sie aus einer festen Liste von Zeichenketten stammen. Statt `cov(<Matrix>, use="pairwise.complete.obs")` ist also auch `cov(<Matrix>, u="pairwise")` als Funktionsaufruf möglich.

1.2.6 Hilfe-Funktionen

R hat ein integriertes Hilfesystem, das auf verschiedene Arten genutzt werden kann: Zum einen ruft `help.start()` eine lokal gespeicherte Webseite auf, von der aus spezifische Hilfe-Seiten erreichbar sind. Zum anderen kann auf diese Seiten mit `help(<Befehlsname>)` zugegriffen werden, in Kurzform auch mit `?<Befehlsname>`. Operatoren müssen bei beiden Versionen in Anführungszeichen gesetzt werden, etwa `"?/"`. Die Hilfe-Funktion lässt sich in RStudio auch über den Help Tab erreichen. Weitere Hinweise zum Hilfesystem liefert `help()` ohne zusätzliche Argumente.

Die Inhalte der Hilfe sind meist knapp und eher technisch geschrieben, zudem setzen sie häufig Vorkenntnisse voraus. Dennoch stellen sie eine wertvolle und reichhaltige Ressource dar, deren Wert sich mit steigender Vertrautheit mit R stärker erschließt. Im Abschnitt `Usage` der Hilfe-Seiten werden verschiedene Anwendungsmöglichkeiten der Funktion beschrieben. Dazu zählen auch unterschiedliche Varianten im Fall von generischen Funktionen, deren Arbeitsweise von der Klasse der übergebenen Argumente abhängt (vgl. Abschn. 15.2.6). Unter `Arguments` wird erläutert, welche notwendigen sowie optionalen Argumente die Funktion besitzt und welcher Wert für ein optionales Argument voreingestellt ist. Der Abschnitt `Value` erklärt, welche Werte die Funktion als Ergebnis zurückliefert. Weiterhin wird die Benutzung der Funktion im Abschnitt `Examples` mit Beispielen erläutert. `example(<Befehlsnahme>)` führt diese Beispiele samt ihrer Ergebnisse auf der Konsole vor.

Wenn der genaue Name einer gesuchten Funktion unbekannt ist, können die Hilfe-Seiten mit `help.search("<Stichwort>")` nach Stichworten gesucht werden. Das Ergebnis führt jene Funktionsnamen auf, in deren Hilfe-Seiten `<Stichwort>` vorhanden ist. Mit `apropos("<Stichwort>")` werden Funktionen ausgegeben, die `<Stichwort>` in ihrem Funktionsnamen tragen. Mit der Funktion `findFn()` aus dem Paket `sos` (Graves, Dorai-Raj & Francois, 2013) lässt sich die Suche nach Funktionen über den installierten Umfang hinaus ausdehnen: Sie durchsucht zusätzlich zum Basisumfang von R alle auf CRAN verfügbaren Zusatzpakete (vgl. Abschn. 1.2.7). Für weiterführende Recherchemöglichkeiten vgl. Abschn. 1.1.5.

1.2.7 Zusatzpakete verwenden

R lässt sich über eigenständig entwickelte Zusatzkomponenten modular erweitern, die in Form von *Paketen* inhaltlich spezialisierte Funktionen zur Datenanalyse mit vorgefertigten Datensätzen und eigener Dokumentation bündeln (Ligges, 2003). Die thematisch geordnete und redaktionell gepflegte Übersicht *Task Views* auf CRAN (Zeileis, 2005) führt dazu etwa unter *Psychometric Models* (Mair, 2014) und *Statistics for the Social Sciences* (Fox, 2014) viele für Psychologen und Sozialwissenschaftler relevante Pakete an. Dort finden sich auch Pakete zu Inhalten, die in diesem Text weitgehend ausgeklammert bleiben, etwa zur Test- und Fragebogenanalyse oder zur Auswertung von Zeitreihen.¹⁶

Während einige Pakete bereits in einer Standardinstallation enthalten sind, aber aus Effizienzgründen erst im Bedarfsfall geladen werden müssen, sind die meisten Zusatzpakete zunächst

¹⁶Der Lesbarkeit halber werden in diesem Buch vorgestellte Pakete nur bei ihrer ersten Verwendung auch zitiert, bei späteren Erwähnungen wird nur ihr Name genannt. Über den im Index markierten Haupteintrag für ein Paket lässt sich die Zitation finden.

manuell zu installieren. Auf Rechnern mit Online-Zugriff lassen sich Zusatzpakete aus RStudio über das Menü Tools: Install Packages... installieren. Gleichermaßen kann in der Konsole mit dem `install.packages()` Befehl die Installation eines Pakets angefordert werden.¹⁷

```
> install.packages(pkgs="Paketname")
```

Der Paketname ist in Anführungszeichen eingeschlossen für das Argument `pkgs` zu nennen. Durch Angabe eines Vektors von Paketnamen (vgl. Abschn. 2.1.1) lassen sich auch mehrere Pakete gleichzeitig installieren: `install.packages(c("Paket 1", "<Paket 2>", ...))`.

Für Rechner ohne Internetanbindung lassen sich die Installationsdateien der Zusatzpakete von einem anderen Rechner mit Online-Zugriff herunterladen, um sie dann manuell auf den Zielrechner übertragen und dort installieren zu können. Um ein Paket auf diese Weise zu installieren, muss von der R-Projektseite kommend einer der CRAN-mirrors und anschließend Contributed extension packages gewählt werden. Die alphabetisch geordnete Liste führt alle verfügbaren Zusatzpakete inkl. einer kurzen Beschreibung auf. Durch Anklicken eines Paketnamens öffnet sich die zugehörige Download-Seite. Sie enthält eine längere Beschreibung sowie u. a. den Quelltext des Pakets (Dateiendung `.tar.gz`), eine zur Installation unter Windows geeignete Archivdatei (Dateiendung `.zip`) sowie Dokumentation im PDF-Format, die u. a. die Funktionen des Zusatzpakets erläutert. Nachdem die Archivdatei auf den Zielrechner übertragen ist, lässt sie sich in RStudio über Tools: Install Packages...: Install from Package Archive File installieren.

R installiert die Zusatzpakete entweder im R-Installationsverzeichnis, oder aber im Heimverzeichnis des Benutzers. Den Pfad zu diesen Verzeichnissen zeigt `.libPaths()` an.¹⁸ Alle installierten Pakete lassen sich in RStudio über das Menü Tools: Check for Package Updates... aktualisieren, sofern eine neue Version auf den CRAN-servern vorhanden ist. Diesem Vorgehen entspricht auf der Konsole der Befehl `update.packages()`.

Damit die Funktionen und Datensätze eines installierten Zusatzpakets auch zur Verfügung stehen, muss es bei jeder neuen R-Sitzung manuell geladen werden. Dafür lassen sich die installierten Zusatzpakete auf der Konsole mit folgenden Befehlen auflisten und laden:¹⁹

```
> installed.packages()                      # installierte Pakete auflisten  
> library(Paketname)                      # Paket laden
```

¹⁷ Die Installation setzt voraus, dass der Benutzer ausreichende Schreibrechte auf dem Computer besitzt, weshalb es u. U. notwendig ist, R zunächst als Administrator zu starten. Mit dem Argument `repos` von `install.packages()` können temporär, mit `setRepositories()` auch dauerhaft andere server als Paketquelle verwendet werden. Hier ist etwa das BioConductor-Projekt (Gentleman, Carey, Bates & others, 2004) mit Paketen vor allem zur Bioinformatik zu nennen. Für die Installation von auf GitHub gehosteten Paketen ist die Funktion `install_github()` aus dem Paket `devtools` (Wickham, 2014b) notwendig. `remove.packages()` deinstalliert ein Paket wieder.

¹⁸ Bei der Installation einer neuen R-Version müssen zuvor manuell hinzugefügte Pakete erneut installiert werden, wenn es sich um einen großen Versionssprung handelt, z. B. von Version 3.0 zu 3.1 – nicht aber von Version 3.1.1 zu 3.1.2. Das Paket-Verzeichnis kann auch frei gewählt werden. Dafür muss eine Textdatei `Renviron.site` im Unterordner `etc/` des R-Programmordners existieren und eine Zeile der Form `R_LIBS="Pfad"` (z. B. `R_LIBS="c:/rlibs"`) mit dem Pfad zu den Paketen enthalten.

¹⁹ Wird versucht, ein nicht installiertes Paket zu laden, erzeugt `library()` einen Fehler. Wenn dagegen das Argument `logical.return=TRUE` gesetzt wird, erzeugt `library()` nur eine Warnung und gibt ein später zur Fallunterscheidung verwendbares `FALSE` zurück (vgl. Abschn. 15.1.1). Auch `require()` warnt nur, wenn ein zu ladendes Paket nicht vorhanden ist.

`installed.packages()` und `library()` zeigen ohne Angabe von Argumenten alle installierten und damit ladbaren Pakete samt ihrer Versionsnummer an. Ist ein Paket geladen, finden sich die in ihm enthaltenen Objekte in einer eigenen Umgebung wieder, die den Namen `package:<Paketname>` trägt, was am von `search()` ausgegebenen Suchpfad zu erkennen ist (vgl. Abschn. 1.2.3).²⁰

Kurzinformationen zu einem ladbaren Paket, etwa die darin enthaltenen Funktionen, liefert `help(package="<Paketname>")`. Viele Pakete bringen darüber hinaus noch ausführlichere Dokumentation im PDF-Format mit, die mit `vignette("<Thema>")` aufgerufen werden kann. Ein Thema kann etwa der Paketname sein, aber auch für Spezialthemen existieren eigene Dokumente. Verfügbare Themen zu den installierten Paketen können durch `vignette()` ohne Angabe von Argumenten angezeigt werden.

Die Ausgabe von `sessionInfo()` zeigt u. a., welche Pakete geladen sind. Ein geladenes Paket kann über `detach(package:<Paketname>)` auch wieder entfernt werden.

Eine Übersicht darüber, welche Datensätze in einem bestimmten Zusatzpaket vorhanden sind, wird mit `data(package="<Paketname>")` geöffnet (vgl. Abschn. 3.2). Diese Datensätze können mit `data(<Datensatz>, package="<Paketname>")` auch unabhängig von den Funktionen des Pakets geladen werden. Ohne Angabe von Argumenten öffnet `data()` eine Liste mit bereits geladenen Datensätzen. Viele Datensätze sind mit einer kurzen Beschreibung ausgestattet, die `help(<Datensatz>)` ausgibt.

1.2.8 Empfehlungen und typische Fehlerquellen

Übersichtlichkeit und Nachvollziehbarkeit sind entscheidende Gesichtspunkte beim Erstellen einer Abfolge von Befehlen, um ihre Korrektheit prüfen und Bausteine der Datenanalyse später wiederverwenden zu können. Befehlssequenzen sollten daher so geschrieben werden, dass sie ein einfaches Verständnis der Vorgänge gewährleisten – etwa durch folgende Maßnahmen:

- Leerzeichen zwischen Befehlsteilen bzw. zwischen Operatoren und Objektnamen verwenden.
- Objekte sinnvoll (inhaltlich aussagekräftig) benennen und dabei ein einheitliches Schema verwenden, z. B. `groupMeans` (*CamelCase*)
- Komplexe Berechnungen in einzelne Schritte aufteilen, deren Zwischenergebnisse separat geprüft werden können.
- Mit dem `#` Zeichen Kommentare einfügen. Kommentare erläutern Befehle und erklären Analysen. Sie dienen anderen Personen dazu, die Bedeutung der Befehle und das Ziel von Auswertungsschritten schnell zu erfassen. Aber auch für den Autor selbst sind Kommentare hilfreich, wenn Befehle längere Zeit nach Erstellen geprüft oder für eine andere Analyse angepasst werden sollen.

²⁰Besitzen verschiedene geladene Pakete Funktionen desselben Namens, maskieren die aus später geladenen Paketen jene aus früher geladenen (vgl. Abschn. 1.3.1). Um explizit auf eine so maskierte Funktion zuzugreifen, ist dem Funktionsnamen der Paketname mit zwei Doppelpunkten voranzustellen, etwa `base::mean()`.

Bei falsch eingegebenen Befehlen bricht R die Auswertung ab und liefert eine Fehlermeldung, die meist einen Rückschluss auf die Ursache erlaubt. Einige typische Fehlerquellen bei der Arbeit mit R sind die folgenden:

- Groß- und Kleinschreibung sind relevant, dies gilt für Objekte, Funktionsnamen und deren Argumente.
- Allgemein sollte bei Fehlern geprüft werden, ob Funktionsnamen und Argumente richtig geschrieben sind. Diese sind in R sehr inkonsistent benannt, so existieren etwa die folgenden Funktionen, deren Namen sich jeweils aus zwei Bestandteilen zusammensetzt: `as.Date()`, `read.table()`, `seq_along()`, `TukeyHSD()`, `zapsmall()`. Zwei Argumente von `read.table()` lauten `row.names` und `colClasses`. Durch die Unterscheidung von Groß- und Kleinschreibung ist das Fehlerpotential hier sehr hoch. Eine Übersicht über die genaue Schreibweise von Argumenten erhält man mit `args(<Funktionsname>)` oder über die Hilfe `?<Funktionsname>`. In grafischen Entwicklungsumgebungen wie RStudio lassen sich Fehler vermeiden, wenn man die Möglichkeiten zur automatischen Tab Vervollständigung angefangener Befehlsnamen verwendet (vgl. Abschn. 1.2.1).
- Das Dezimaltrennzeichen ist immer der Punkt (2.173), nicht das Komma.
- Mehrere Argumente von Funktionen sind durch ein Komma voneinander zu trennen.
- Zeichenketten müssen fast immer in Anführungszeichen stehen.
- Alle öffnenden Klammern müssen auch geschlossen werden, dabei ist besonders auf die richtige Position der schließenden Klammer und auf den richtigen Typ (eckig oder rund) zu achten. Auch hier hilft eine Entwicklungsumgebung wie RStudio, die für jede geöffnete Klammer automatisch eine schließende einfügt und ein Klammernpaar farblich hervorhebt.

1.3 Datenstrukturen: Klassen, Objekte, Datentypen

Die Datenstrukturen, die in R Informationen repräsentieren, sind im wesentlichen eindimensionale Vektoren (`vector`), zweidimensionale Matrizen (`matrix`), verallgemeinerte Matrizen mit auch mehr als zwei Dimensionen (`array`), Listen (`list`), Datensätze (`data.frame`) und Funktionen (`function`). Die gesammelten Eigenschaften jeweils einer dieser Datenstrukturen werden als *Klasse* bezeichnet (für Details vgl. Chambers, 2008; Ligges, 2014).

Daten werden in R in benannten Objekten gespeichert. Jedes Objekt ist eine konkrete Verkörperung (*Instanz*) einer der o. g. Klassen, die Art und Struktur der im Objekt gespeicherten Daten festlegt. Die Klasse eines Objekts kann mit dem Befehl `class(<Objekt>)` erfragt werden, wobei als Klasse von Vektoren der Datentyp der in ihm gespeicherten Werte gilt, z. B. `numeric` (s. u.). Die Funktionen, deren Namen nach dem Muster `is.<Klasse>(<Objekt>)` aufgebaut sind, prüfen, ob ein vorliegendes Objekt von einer gewissen Klasse ist. So gibt etwa `is.matrix(<Objekt>)` an, ob `<Objekt>` die Klasse `matrix` hat (Ausgabe TRUE) oder nicht (Ausgabe FALSE).

Bestehende Objekte einer bestimmten Klasse können unter gewissen Voraussetzungen in Objekte einer anderen Klasse konvertiert werden. Zu diesem Zweck stellt R eine Reihe von Funktionen

bereit, deren Namen nach dem Schema `as.<Klasse>(<Objekt>)` aufgebaut sind. Um ein Objekt in einen Vektor umzuwandeln, wäre demnach `as.vector(<Objekt>)` zu benutzen. Mehr Informationen zu diesem Thema finden sich bei der Behandlung der einzelnen Klassen.

Intern werden die Daten vieler Objekte durch einen Vektor, d. h. durch eine linear geordnete Menge einzelner Werte repräsentiert. Jedes Objekt besitzt eine Länge, die meist der Anzahl der im internen Vektor gespeicherten Elemente entspricht und durch den Befehl `length(<Objekt>)` abgefragt werden kann.

Objekte besitzen darüber hinaus einen Datentyp (*Modus*), der sich auf die Art der im Objekt gespeicherten Informationen bezieht und mit `mode(<Objekt>)` ausgegeben werden kann – unterschieden werden vornehmlich numerische, alphanumerische und logische Modi (vgl. Abschn. 1.3.5). Ein Objekt der Klasse `matrix` könnte also z. B. mehrere Wahrheitswerte speichern und somit den Datentyp `logical` besitzen.

Ein Objekt kann zudem *Attribute* aufweisen, die zusätzliche Informationen über die in einem Objekt enthaltenen Daten speichern. Sie können mit dem Befehl `attributes(<Objekt>)` und `attr(<Objekt>, which=<Attribut>)` abgefragt sowie über `attr()` auch geändert werden.²¹ Meist versieht R von sich aus Objekte mit Attributen, so kann etwa die Klasse eines Objekts im Attribut `class` gespeichert sein. Man kann Attribute aber auch selbst i. S. einer freien Beschreibung nutzen, die man mit einem Objekt assoziieren möchte – hierfür eignet sich alternativ auch `comment()`.

Über die interne Struktur eines Objekts, also seine Zusammensetzung aus Werten samt ihrer Datentypen, gibt `str(<Objekt>)` Auskunft.

1.3.1 Objekte benennen

Objekte tragen i. d. R. einen Namen (ihr *Symbol*) beliebiger Länge, über den sie in Befehlen identifiziert werden. Objektnamen sollten mit einem Buchstaben beginnen, können aber ab der zweiten Stelle neben Buchstaben auch Ziffern, Punkte und Unterstriche enthalten. Von der Verwendung anderer Sonderzeichen wie auch von deutschen Umlauten ist abzuraten, selbst wenn dies bisweilen möglich ist.²² Groß- und Kleinschreibung werden bei Objektnamen und Befehlen unterschieden, so ist das Objekt `asdf` ein anderes als `Asdf`. Objekte dürfen nicht den Namen spezieller Schlüsselwörter wie `if` tragen, die in der Hilfe-Seite `?Reserved` aufgeführt sind.

Ebenso sollten keine Objekte mit Namen versehen werden, die gleichzeitig Funktionen in R bezeichnen, selbst wenn dies möglich ist. Kommt es dennoch zu einer *Maskierung* von bereits durch R vergebenen Namen durch selbst angelegte Objekte, ist dies i. d. R. unproblematisch. Dies liegt daran, dass es nicht nur einen, sondern mehrere workspaces als voneinander abgegrenzte Einheiten gibt, die Objekte desselben Namens beinhalten können, ohne dass diese sich wechselseitig überschreiben würden (vgl. Abschn. 1.2.3). Ob Namenskonflikte, also mehrfach vergebene Objektnamen, vorliegen, kann mit `conflicts(detail=TRUE)` geprüft werden. `exists("⟨Name⟩")` gibt an, ob `⟨Name⟩` schon als Symbol verwendet wird.

²¹ Mit `structure()` lassen sich auch mehrere Attribute gleichzeitig setzen.

²² Wenn ein Objektname dennoch nicht zulässige Zeichen enthält, kann man nichtsdestotrotz auf das Objekt zugreifen, indem man den Namen in rückwärts gerichtete Hochkommata setzt (`⟨Objektname⟩`).

1.3.2 Zuweisungen an Objekte

Um Ergebnisse von Berechnungen zu speichern und wiederzuverwenden zu können, müssen diese einem benannten Objekt zugewiesen werden. Objekte können dabei einzelne Zahlen aufnehmen, aber auch Text oder andere komplexe Inhalte haben. Zuweisungen, z. B. an ein Objekt `x1`, können auf zwei gängige Arten geschehen:

```
> x1 <- 4.5  
> x1 = 4.5
```

Zur Vermeidung von Mehrdeutigkeiten bei komplizierteren Eingaben sollte die erste Methode mit `<-` bevorzugt werden. Das vielleicht vertrautere `=` sollte der Zuweisung von Funktionsargumenten vorbehalten bleiben (vgl. Abschn. 1.2.5), um die Richtung der Zuweisung eindeutig zu halten. Im Fall des `<-` Operators erfolgt die Zuweisung von rechts nach links in Richtung des Pfeils und kann sich auch über mehrere Objekte erstrecken:

```
> x2 <- x3 <- 10  
> x2  
[1] 10  
  
> x3  
[1] 10
```

Objekte können in Befehlen genauso verwendet werden, wie die Daten, die in ihnen gespeichert sind, d. h. Objektnamen stehen in Berechnungen für die im Objekt gespeicherten Werte.

```
> x1 * 2  
[1] 9  
  
> x1^x1 - x2  
[1] 859.874
```

1.3.3 Objekte ausgeben

Bei Zuweisungen zu Objekten gibt R den neuen Wert nicht aus, der letztlich im Zielobjekt gespeichert wurde. Um sich den Inhalt eines Objekts anzeigen zu lassen, gibt es folgende Möglichkeiten:

```
> print(x1)          # print(<Objektname>) Funktion  
> get("x1")         # get("<Objektname>") Funktion  
> x1                # Objektnamen nennen - ruft implizit print() auf  
> (x1 <- 4.5)        # Befehl in runde Klammern setzen - zeigt nur  
                      # die durch den Befehl veränderten Werte an
```

Es ist allgemein dazu zu raten, häufig runde Klammern um einen Befehl zu setzen. Dadurch können die in Zwischenrechnungen veränderten Werte mit ausgegeben werden, was die Kontrolle der Richtigkeit einzelner Arbeitsschritte erleichtert. Die Variante `get("<Objektname>")` eignet sich besonders für Situationen, in denen der Name des auszugebenden Objekts nur als

Zeichenkette in einem anderen Objekts gespeichert ist und deshalb nicht von Hand eingetippt werden kann.²³

```
> varName <- "x1"      # Variable, die Objektnamen enthält  
> get(varName)        # gewünschtes Objekt ausgeben  
[1] 4.5
```

Um mit Zwischenergebnissen weiterzurechnen, sollten diese nicht auf der Konsole ausgegeben, dort abgelesen und später von Hand als fester Wert in einer Rechnung eingesetzt werden. Dies würde zum einen dazu führen, dass die Genauigkeit beim Rechnen mit Dezimalzahlen unnötig auf die Anzahl ausgegebener Dezimalstellen begrenzt wird. Zum anderen verhindert ein solches Vorgehen, dass die erstellten Befehle auf neue Situationen übertragbar sind, in denen sich nicht exakt dasselbe Zwischenergebnis einstellt. Stattdessen sollten Zwischenergebnisse immer einem eigenen Objekt zugewiesen werden, das dann in späteren Rechnungen auftauchen kann.

Wurde vergessen, das Ergebnis eines Rechenschritts als Objekt zu speichern, so lässt sich das letzte ausgegebene Ergebnis mit `.Last.value` erneut anzeigen und einem Objekt zuweisen.

1.3.4 Objekte anzeigen lassen, umbenennen und entfernen

Um sich einen Überblick über alle im workspace vorhandenen Objekte zu verschaffen, dient `ls()` (`list`). Objekte, deren Name mit einem Punkt beginnt, sind dabei versteckt – sie werden erst mit `ls(all.names=TRUE)` angezeigt.

```
> ls()  
[1] "x1" "x2" "x3"
```

Um Objekte umzubenennen gibt es zwei Möglichkeiten: Zum einen kann das alte Objekt wie beschrieben einem neuen, passend benannten Objekt zugewiesen werden. Ergibt sich der gewünschte Name dagegen aus dem Inhalt einer Variable und kann deshalb nicht von Hand eingetippt werden, ist es mit `assign("<Name>", value=<Objekt>")` möglich, einem neuen Objekt den Inhalt eines bereits bestehenden Objekts `value` zuzuweisen.

```
> newNameVar <- "varNew"      # Variable, die neuen Objektnamen enthält  
> assign(newNameVar, x1)       # weise neuem Objekt Wert von x1 zu  
> varNew                      # Objekt mit neuem Namen  
[1] 4.5
```

Vorhandene Objekte können mit `rm(<Objekt>)` (`remove`) gelöscht werden. Sollen alle bestehenden Objekte entfernt werden, kann dies mit dem Befehl `rm(list=ls(all.names=TRUE))` oder in RStudio über das Besen-icon Clear im Environment Tab geschehen.

```
> age <- 22  
> rm(age); age  
Fehler: Objekt "age" nicht gefunden
```

²³Um analog Objekte mit einem später festgelegten Namen zu erstellen, vgl. Abschn. 1.3.4.

1.3.5 Datentypen

Der Datentyp eines Objekts bezieht sich auf die Art der in ihm gespeicherten Informationen und lässt sich mit `mode(<Objekt>)` ausgeben. Neben den in Tab. 1.2 aufgeführten Datentypen existieren noch weitere, über die `?mode` Auskunft gibt.

Tabelle 1.2: Datentypen

Beschreibung	Beispiel	Datentyp
leere Menge	NULL	NULL
logische Werte	TRUE, FALSE (T, F)	logical
ganze und reelle Zahlen	3.14	numeric ²⁴
komplexe Zahlen	3.14 + 1i	complex
Buchstaben und Zeichenfolgen (immer in Anführungszeichen einzugeben) ²⁵	"Hello"	character

```
> charVar <- "asdf"
> mode(charVar)
[1] "character"
```

Die Funktionen, deren Namen nach dem Muster `is.<Datentyp>(<Objekt>)` aufgebaut sind, prüfen, ob ein Objekt Werte von einem bestimmten Datentyp speichert. So gibt etwa die Funktion `is.logical(<Objekt>)` an, ob die Werte in `<Objekt>` vom Datentyp `logical` sind.

So wie Objekte einer bestimmten Klasse in Objekte einer anderen Klasse umgewandelt werden können, lässt sich auch der Datentyp der in einem Objekt gespeicherten Werte in einen anderen konvertieren. Die Funktionen zur Umwandlung des Datentyps sind nach dem Muster `as.<Datentyp>(<Objekt>)` benannt. Um etwa eine Zahl in den zugehörigen Text umzuwandeln, ist der Befehl `as.character(<Zahl>)` zu benutzen.

```
> is.character(1.23)
[1] FALSE

> as.character(1.23)
[1] "1.23"
```

²⁴Für reelle Zahlen (`numeric`) existieren u. a. zwei Möglichkeiten, sie in einem Computer intern zu repräsentieren: Ganze Zahlen können mit einem L hinter der Zahl gekennzeichnet werden (für `long integer`, z. B. `5L`), wodurch R sie dann auch als solche speichert (`integer`). Andernfalls werden alle Zahlen in R als Gleitkommazahlen mit doppelter Genauigkeit gespeichert (`double`). Dies lässt sich mit dem Befehl `typeof(<Objekt>)` abfragen. Ob ein Objekt einen bestimmten Speichertyp aufweist, wird mit Funktionen der `is.<Speicherart>(<Objekt>)` Familie geprüft (z. B. `is.double()`). Weitere Angaben zur internen Implementierung von Zahlen und den daraus resultierenden Beschränkungen gibt `.Machine` aus, etwa die größtmögliche ganze Zahl `.Machine$integer.max` oder die kleinste positive Gleitkommazahl, die noch von 0 unterscheidbar ist `.Machine$double.eps`.

²⁵Dies können einfache ('<Zeichen>') oder doppelte ("<Zeichen>") Anführungszeichen sein. Innerhalb einfacher Anführungszeichen können auch Zeichenketten stehen, die ihrerseits doppelte Anführungszeichen beinhalten ('a"b'), während diese innerhalb doppelter Anführungszeichen als *Escape-Sequenz* mit vorangestelltem backslash zu schreiben sind ("a\"b", vgl. `?Quotes`).

```
> as.logical(2)
[1] TRUE
```

Bei der Umwandlung von Datentypen besteht eine Hierarchie entsprechend der in Tab. 1.2 aufgeführten Reihenfolge. Weiter unten stehende Datentypen können Werte aller darüber stehenden Datentypen ohne Informationsverlust repräsentieren, nicht jedoch umgekehrt: Jede reelle Zahl lässt sich z. B. genauso gut als komplexe Zahl mit imaginärem Anteil 0 speichern (1.23 ist gleich $1.23 + 0i$), jeder logische Wert entsprechend einer bestimmten Konvention als ganze Zahl (TRUE entspricht der 1, FALSE der 0). Umgekehrt jedoch würden viele unterschiedliche komplexe Zahlen nur als gleiche reelle Zahl gespeichert werden können, und viele unterschiedliche ganze Zahlen würden als gleicher logischer Wert repräsentiert (alle Zahlen ungleich 0 als TRUE, die 0 als FALSE).

Während sich alle Zahlen mit `as.character(<Zahl>)` in die zugehörige Zeichenkette umwandeln lassen, ist dies umgekehrt nicht allgemein möglich. `as.numeric("<Text>")` ergibt nur für Zeichenketten der Form "`<Zahl>`" den entsprechenden numerischen Wert, andernfalls NA als Konstante, die für einen fehlenden Wert steht (vgl. Abschn. 2.11). Analog führt `as.logical("abc")` zum Ergebnis NA, während `as.logical("TRUE")` und `as.logical("FALSE")` in TRUE bzw. FALSE umwandelbar sind.

1.3.6 Logische Werte, Operatoren und Verknüpfungen

Das Ergebnis eines logischen Vergleichs mit den in Tab. 1.3 genannten Operatoren sind Wahrheitswerte, die entweder WAHR (TRUE) oder FALSCH (FALSE) sein können.²⁶ Wahrheitswerte lassen sich auch in numerischen Rechnungen nutzen, dem Wert TRUE entspricht dann die 1, dem Wert FALSE die 0.

Tabelle 1.3: Logische Operatoren, Funktionen und Konstanten

Operator / Funktion / Konstante	Beschreibung
<code>!=, ==</code>	Vergleich: ungleich, gleich
<code>>, >=, <, <=</code>	Vergleich: größer, größer-gleich, kleiner, kleiner-gleich
<code>!</code>	logisches NICHT (Negation)
<code>&, &&</code>	Verknüpfung: logisches UND
<code> , </code>	Verknüpfung: logisches ODER (einschließend)
<code>xor()</code>	logisches ENTWEDER-ODER (ausschließend)
TRUE, FALSE (T, F)	logische Wahrheitswerte: WAHR, FALSCH (abgekürzt)

```
> TRUE == TRUE      > TRUE == FALSE     > !TRUE           > !FALSE
[1] TRUE            [1] FALSE          [1] FALSE          [1] TRUE

> TRUE != TRUE     > TRUE != FALSE
[1] FALSE           [1] TRUE
```

²⁶Für Hilfe zu diesem Thema vgl. `?base::Logic`.

```
> TRUE & TRUE      > TRUE & FALSE     > FALSE & FALSE    > FALSE & TRUE
[1] TRUE           [1] FALSE        [1] FALSE        [1] FALSE

> TRUE | TRUE      > TRUE | FALSE     > FALSE | FALSE    > FALSE | TRUE
[1] TRUE           [1] TRUE         [1] FALSE        [1] TRUE

> 4 < 8            > 7 < 3          > 4 > 4          > 4 >= 4
[1] TRUE           [1] FALSE        [1] FALSE        [1] TRUE
```

Statt des logischen Vergleichsoperators `==` kann zum Prüfen zweier Objekte `x` und `y` auf exakte Gleichheit auch `identical(x, y)` eingesetzt werden. Dabei ist zu beachten, dass `identical()` anders als `==` die Gleichheit nur dann bestätigt, wenn `x` und `y` in R auch intern auf dieselbe Weise repräsentiert sind, etwa denselben Datentyp besitzen.

```
# FALSE, da 4L ganzzahlig, 4 aber als Gleitkommazahl repräsentiert ist
> identical(4L, 4)
[1] FALSE

> 4L == 4          # TRUE, da bezeichneter Wert identisch
[1] TRUE
```

Die Funktion `isTRUE(<Objekt>)` prüft komplexere Objekte darauf, ob sie dem Wert `TRUE` entsprechen. Ihr Ergebnis ist in jedem Fall entweder `TRUE` oder `FALSE`. Auch Objekte, die möglicherweise Werte mit undefiniertem Wahrheitsstatus enthalten, lassen sich so eindeutig prüfen, ob sie `TRUE` sind. Dazu zählen fehlende Werte (`NA`, vgl. Abschn. 2.11), `NaN`, `NULL`, sowie Vektoren der Länge 0. Dies ist insbesondere bei Fallunterscheidungen mit `if()` relevant (vgl. Abschn. 15.1.1).

```
> NaN == TRUE
[1] NA

> NULL == TRUE
logical(0)

> isTRUE(NaN)
[1] FALSE

> isTRUE(NULL)
[1] FALSE
```

Demgegenüber bestätigt die `all.equal()` Funktion die Gleichheit zweier Objekte auch dann, wenn sie sich leicht unterscheiden. Ihre Verwendung ist dann zu empfehlen, wenn entschieden werden soll, ob zwei Dezimalzahlen denselben Wert haben.

```
> all.equal(target=<Objekt1>, current=<Objekt2>,
+           tolerance=<relative Abweichung>, check.attributes=TRUE)
```

Für `target` und `current` sind die zu vergleichenden Objekte zu nennen. In der auf `TRUE` gesetzten Voreinstellung für `check.attributes` berücksichtigt die Prüfung auf Gleichheit auch

die Attribute eines Objekts, zu denen insbesondere die Benennungen einzelner Werte zählen (vgl. Abschn. 1.3, 2.1.4). Die Objekte `target` und `current` gelten auch dann als gleich, wenn ihre Werte nur ungefähr, d. h. mit einer durch `tolerance` festgelegten Genauigkeit übereinstimmen. Aufgrund der Art, in der Computer Gleitkommazahlen intern speichern und verrechnen, sind kleine Abweichungen in Rechenergebnissen nämlich schon bei harmlos wirkenden Ausdrücken möglich. So ergibt der Vergleich `0.1 + 0.2 == 0.3` fälschlicherweise `FALSE` und `1 %/% 0.1` ist `9` statt `10`. `sin(pi)` wird als `1.224606e-16` und nicht exakt `0` berechnet, ebenso ist `1-((1/49)*49)` nicht exakt `0`, sondern `1.110223e-16`. Dagegen ist `1-((1/48)*48)` exakt `0`.²⁷ Dies sind keine R-spezifischen Probleme, sie können nicht allgemein verhindert werden (Cowlishaw, 2008; Goldberg, 1991).

Allerdings liefert `all.equal()` im Fall der Ungleichheit nicht den Wahrheitswert `FALSE` zurück, sondern ein Maß der relativen Abweichung. Wird ein einzelner Wahrheitswert als Ergebnis benötigt, muss `all.equal()` deshalb mit `isTRUE()` verschachtelt werden.

```
> isTRUE(all.equal(0.123450001, 0.123450000))
[1] TRUE

> 0.123400001 == 0.123400000
[1] FALSE

> all.equal(0.12345001, 0.12345000)
[1] "Mean relative difference: 8.100445e-08"

> isTRUE(all.equal(0.12345001, 0.12345000))
[1] FALSE
```

²⁷Tauchen sehr kleine Zahlen, die eigentlich `0` sein sollten, zusammen mit größeren Zahlen in einem Ergebnis auf, eignet sich `zapsmall()`, um sie i. S. einer besseren Übersichtlichkeit auch tatsächlich als `0` ausgeben zu lassen.

Kapitel 2

Elementare Dateneingabe und -verarbeitung

Die folgenden Abschnitte sollen gleichzeitig die grundlegenden Datenstrukturen in R sowie Möglichkeiten zur deskriptiven Datenauswertung erläutern. Die Reihenfolge der Themen ist dabei so gewählt, dass die abwechselnd vorgestellten Datenstrukturen und darauf aufbauenden deskriptiven Methoden nach und nach an Komplexität gewinnen.

2.1 Vektoren

R ist eine vektorbasierte Sprache, ist also auf die Verarbeitung von in Vektoren angeordneten Daten ausgerichtet. Ein Vektor ist dabei lediglich eine Datenstruktur für eine sequentiell geordnete Menge einzelner Werte und nicht mit dem mathematischen Konzept eines Vektors zu verwechseln. Da sich empirische Daten einer Variable meist als eine linear anzuordnende Wertemenge betrachten lassen, sind Vektoren als Organisationsform gut für die Datenanalyse geeignet. Vektoren sind in R die einfachste Datenstruktur für Werte, d. h. auch jeder Skalar oder andere Einzelwert ist ein Vektor der Länge 1.

2.1.1 Vektoren erzeugen

Vektoren werden durch Funktionen erzeugt, die den Namen eines Datentyps tragen und als Argument die Anzahl der zu speichernden Elemente erwarten, also etwa `numeric(<Anzahl>)`.¹ Die Elemente des Vektors werden hierbei auf eine Voreinstellung gesetzt, die vom Datentyp abhängt – 0 für `numeric()`, "" für `character` und FALSE für `logical`.

```
> numeric(4)
[1] 0 0 0 0

> character(2)
[1] "" ""
```

Als häufiger genutzte Alternative lassen sich Vektoren auch mit der Funktion `c(<Wert1>, <Wert2>, ...)` erstellen (*concatenate*), die die Angabe der zu speichernden Werte benötigt. Ein das Alter von sechs Personen speichernder Vektor könnte damit so erstellt werden:

¹Ein leerer Vektor entsteht analog, z. B. durch `numeric(0)`. Auf 32bit-Systemen kann ein Vektor höchstens `.Machine$integer.max` viele $(2^{31} - 1)$ Elemente enthalten. Diese Beschränkung wurde mit R-Version 3.0.0 auf 64bit-Systemen aufgehoben. Mit `vector(mode="Klasse", n=(Länge))` lassen sich beliebige Objekte der für `mode` genannten Klasse der Länge `n` erzeugen.

```
> (age <- c(18, 20, 30, 24, 23, 21))
[1] 18 20 30 24 23 21
```

Dabei werden die Werte in der angegebenen Reihenfolge gespeichert und intern mit fortlaufenden Indizes für ihre Position im Vektor versehen. Sollen bereits bestehende Vektoren zusammengefügt werden, ist ebenfalls `c()` zu nutzen, wobei statt eines einzelnen Wertes auch der Name eines bereits bestehenden Vektors angegeben werden kann.

```
> addAge <- c(27, 21, 19)                      # zusätzlicher Vektor
> (ageNew <- c(age, addAge))                  # kombinierter Vektor
[1] 18 30 30 25 23 21 27 21 19
```

Mit `length(<Vektor>)` wird die Länge eines Vektors, d. h. die Anzahl der in ihm gespeicherten Elemente, erfragt.

```
> length(age)
[1] 6
```

Auch Zeichenketten können die Elemente eines Vektors ausmachen. Dabei zählt die leere Zeichenkette `""` ebenfalls als ein Element.

```
> (chars <- c("lorem", "ipsum", "dolor", ""))
[1] "lorem" "ipsum" "dolor" ""
```

```
> length(chars)
[1] 4
```

Zwei aus Zeichen bestehende Vektoren sind in R bereits vordefiniert, `LETTERS` und `letters`, die jeweils alle Buchstaben A–Z bzw. a–z in alphabetischer Reihenfolge als Elemente besitzen.

```
> LETTERS[c(1, 2, 3)]                         # Alphabet in Großbuchstaben
[1] "A" "B" "C"

> letters[c(4, 5, 6)]                          # Alphabet in Kleinbuchstaben
[1] "d" "e" "f"
```

2.1.2 Elemente auswählen und verändern

Um ein einzelnes Element eines Vektors abzurufen, wird seine Position im Vektor (sein Index) in eckigen Klammern, dem `[<Index>]` Operator, hinter dem Objektnamen angegeben.² Indizes beginnen bei 1 für die erste Position³ und enden bei der Länge des Vektors. Werden größere Indizes verwendet, erfolgt als Ausgabe die für einen fehlenden Wert stehende Konstante `NA` (vgl. Abschn. 2.11).

²Für Hilfe zu diesem Thema vgl. `?Extract`. Auch der Index-Operator ist eine Funktion, kann also gleichermaßen in der Form `"["(<Vektor>, <Index>)` verwendet werden (vgl. Abschn. 1.2.5, Fußnote 14).

³Dies mag selbstverständlich erscheinen, in anderen Sprachen wird jedoch oft der Index 0 für die erste Position und allgemein der Index $i - 1$ für die i -te Position verwendet. Für einen Vektor `x` ist das Ergebnis von `x[0]` immer ein leerer Vektor mit demselben Datentyp wie jener von `x`.

```
> age[4] # 4. Element von age
[1] 24

> (ageLast <- age[length(age)]) # letztes Element von age
[1] 21

> age[length(age) + 1]
[1] NA
```

Ein Vektor muss nicht unbedingt einem Objekt zugewiesen werden, um indiziert werden zu können, dies ist auch für unbenannte Vektoren möglich.

```
> c(11, 12, 13, 14)[2]
[1] 12
```

Mehrere Elemente eines Vektors lassen sich gleichzeitig abrufen, indem ihre Indizes in Form eines Indexvektors in die eckigen Klammern eingeschlossen werden. Dazu kann man zunächst einen eigenen Vektor erstellen, dessen Name dann in die eckigen Klammern geschrieben wird. Ebenfalls kann der Befehl zum Erzeugen eines Vektors direkt in die eckigen Klammern verschachtelt werden. Der Indexvektor kann auch länger als der indizierte Vektor sein, wenn einzelne Elemente mehrfach ausgegeben werden sollen. Das Weglassen eines Index mit `<Vektor>[]` führt dazu, dass alle Elemente des Vektors ausgegeben werden.

```
> idx <- c(1, 2, 4)
> age[idx]
[1] 18 20 24

> age[c(3, 5, 6)]
[1] 30 23 21

> age[c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6)]
[1] 18 18 20 20 30 30 24 24 23 23 21 21
```

Beinhaltet der Indexvektor fehlende Werte (`NA`), erzeugt dies in der Ausgabe ebenfalls einen fehlenden Wert an der entsprechenden Stelle.

```
> age[c(4, NA, 1)]
[1] 25 NA 17
```

Wenn alle Elemente bis auf ein einzelnes abgerufen werden sollen, ist dies am einfachsten zu erreichen, indem der Index des nicht erwünschten Elements mit negativem Vorzeichen in die eckigen Klammern geschrieben wird.⁴ Sollen mehrere Elemente nicht ausgegeben werden, verläuft der Aufruf analog zum Aufruf gewünschter Elemente, wobei mehrere Variationen mit dem negativen Vorzeichen möglich sind.

⁴Als Indizes dürfen in diesem Fall keine fehlenden Werte (`NA`) vorkommen, ebenso darf der Indexvektor nicht leer sein.

```
> age[-3]                      # alle Elemente bis auf das 3.
[1] 18 20 24 23 21

> age[c(-1, -2, -4)]          # alle Elemente bis auf das 1., 2. und 4.
[1] 30 23 21

> age[-c(1, 2, 4)]            # alle Elemente bis auf das 1., 2. und 4.
[1] 30 23 21

> age[-idx]                   # alle Elemente bis auf die Indizes im Vektor idx
[1] 30 23 21
```

Die in einem Vektor gespeicherten Werte können nachträglich verändert werden. Dazu muss der Position des zu ändernden Wertes der neue Wert zugewiesen werden.

```
> age[4] <- 25
> age
[1] 18 20 30 25 23 21
```

Das Verändern von mehreren Elementen gleichzeitig geschieht analog. Dazu lassen sich die Möglichkeiten zur Auswahl mehrerer Elementen nutzen und diesen in einem Arbeitsschritt neue Werte zuweisen. Dabei müssen die zugewiesenen Werte ebenfalls durch einen Vektor repräsentiert sein. Fehlt bei Zuweisungen der Index $\langle\text{Vektor}\rangle[]$, werden alle Elemente des Vektors ersetzt. Wenn der zugewiesene Vektor dabei weniger Elemente als der veränderte Vektor besitzt, wird er automatisch passend verlängert (vgl. Abschn. 2.5.4).

```
> age[idx] <- c(17, 30, 25)
> age
[1] 17 30 30 25 23 21

# alle Elemente gleichzeitig ersetzen mit zyklischer Verlängerung
> age[] <- c(1, 2)
> age
[1] 1 2 1 2 1 2
```

Um Vektoren zu verlängern, also mit neuen Elementen zu ergänzen, kann zum einen der $\langle\text{Index}\rangle$ Operator benutzt werden, wobei als Index nicht belegte Positionen angegeben werden.⁵ Zum anderen kann auch hier $c(\langle\text{Wert1}\rangle, \langle\text{Wert2}\rangle, \dots)$ Verwendung finden. Als Alternative steht die `append(<Vektor>, values=<Vektor>)` Funktion zur Verfügung, die an einen Vektor die Werte eines unter `values` genannten Vektors anhängt.

```
> charVec1 <- c("Z", "Y", "X")
> charVec1[c(4, 5, 6)] <- c("W", "V", "U")
> charVec1
[1] "Z" "Y" "X" "W" "V" "U"
```

⁵Bei der Verarbeitung sehr großer Datenmengen ist zu bedenken, dass die schrittweise Vergrößerung von Objekten aufgrund der dafür notwendigen internen Kopiervorgänge ineffizient ist. Objekte sollten deshalb bevorzugt bereits mit der Größe und dem Datentyp angelegt werden, die sie später benötigen.

```
> (charVec2 <- c(charVec1, "T", "S", "R"))
[1] "Z" "Y" "X" "W" "V" "U" "T" "S" "R"

> (charVec3 <- append(charVec2, c("Q", "P", "O")))
[1] "Z" "Y" "X" "W" "V" "U" "T" "S" "R" "Q" "P" "O"
```

2.1.3 Datentypen in Vektoren

Vektoren können Werte unterschiedlicher Datentypen speichern, etwa `numeric`, wenn sie Zahlen beinhalten, oder `character` im Fall von Zeichenketten. Letztere müssen dabei immer in Anführungszeichen stehen. Jeder Vektor kann aber nur einen Datentyp besitzen, alle seine Elemente haben also denselben Datentyp. Fügt man einem numerischen Vektor eine Zeichenkette hinzu, so werden seine numerischen Elemente automatisch in Zeichenketten umgewandelt,⁶ was man an den hinzugekommenen Anführungszeichen erkennt und mit `mode(Vektor)` überprüfen kann.

```
> charVec4 <- "word"
> numVec   <- c(10, 20, 30)
> (combVec <- c(charVec4, numVec))
[1] "word" "10" "20" "30"

> mode(combVec)
[1] "character"
```

2.1.4 Elemente benennen

Es ist möglich, die Elemente eines Vektors bei seiner Erstellung zu benennen. Die Elemente können dann nicht nur über ihren Index, sondern auch über ihren in Anführungszeichen gesetzten Namen angesprochen werden.⁷ In der Ausgabe wird der Name eines Elements in der über ihm stehenden Zeile mit aufgeführt.

```
> (namedVec1 <- c(elem1="first", elem2="second"))
elem1     elem2
"first"   "second"

> namedVec1["elem1"]
elem1
"first"
```

⁶Allgemein gesprochen werden alle Elemente in den umfassendsten Datentyp umgewandelt, der notwendig ist, um alle Werte ohne Informationsverlust zu speichern (vgl. Abschn. 1.3.5).

⁷Namen werden als Attribut gespeichert und sind mit `attributes(Vektor)` sichtbar (vgl. Abschn. 1.3). Elemente lassen sich über den Namen nur auswählen, nicht aber mittels `(Vektor)[-<Name>]` ausschließen, hierfür bedarf es des numerischen Index.

Auch im nachhinein lassen sich Elemente benennen, bzw. vorhandene Benennungen ändern – beides geschieht mit `names(⟨Vektor⟩)`.

```
> (namedVec2 <- c(val1=10, val2=-12, val3=33))
val1  val2  val3
 10    -12    33

> names(namedVec2)                                # vorhandene Namen
[1] "val1" "val2" "val3"

> names(namedVec2) <- c("A", "B", "C")           # verändere Namen
> namedVec2
 A      B      C
10    -12    33
```

2.1.5 Elemente löschen

Elemente eines Vektors lassen sich nicht im eigentlichen Sinne löschen. Denselben Effekt kann man stattdessen über zwei mögliche Umwege erzielen. Zum einen kann ein bestehender Vektor mit einer Auswahl seiner eigenen Elemente überschrieben werden.

```
> vec <- c(10, 20, 30, 40, 50)
> vec <- vec[-c(4, 5)]
> vec
[1] 10 20 30
```

Zum anderen kann ein bestehender Vektor über `length()` verkürzt werden, indem ihm eine Länge zugewiesen wird, die kleiner als seine bestehende ist. Gelöscht werden dabei die überzähligen Elemente am Ende des Vektors.

```
> vec          <- c(1, 2, 3, 4, 5)
> length(vec) <- 3
> vec
[1] 1 2 3
```

2.2 Logische Operatoren

Verarbeitungsschritte mit logischen Vergleichen und Werten treten häufig bei der Auswahl von Teilmengen von Daten sowie bei der Recodierung von Datenwerten auf. Dies liegt vor allem an der Möglichkeit, in Vektoren und anderen Datenstrukturen gespeicherte Werte auch mit logischen Indexvektoren auszuwählen (vgl. Abschn. 2.2.2).

2.2.1 Vektoren mit logischen Operatoren vergleichen

Vektoren werden oft mit Hilfe logischer Operatoren mit einem bestimmten Wert, oder auch mit anderen Vektoren verglichen um zu prüfen, ob die Elemente gewisse Bedingungen erfüllen. Als Ergebnis der Prüfung wird ein logischer Vektor mit Wahrheitswerten ausgegeben, der die Resultate der elementweisen Anwendung des Operators beinhaltet.

Als Beispiel seien im Vektor `age` wieder die Daten von sechs Personen gespeichert. Zunächst sollen jene Personen identifiziert werden, die jünger als 24 Jahre sind. Dazu wird der `<` Operator verwendet, der als Ergebnis einen Vektor mit Wahrheitswerten liefert, der für jedes Element separat angibt, ob die Bedingung < 24 zutrifft. Andere Vergleichsoperatoren, wie gleich (`==`), ungleich (`!=`), etc. funktionieren analog.

```
> age <- c(17, 30, 30, 24, 23, 21)
> age < 24
[1] TRUE FALSE FALSE FALSE TRUE TRUE
```

Wenn zwei Vektoren miteinander logisch verglichen werden, wird der Operator immer auf ein zueinander gehörendes Wertepaar angewendet, also auf Werte, die sich an derselben Position in ihrem jeweiligen Vektor befinden.

```
> x <- c(2, 4, 8)
> y <- c(3, 4, 5)
> x == y
[1] FALSE TRUE FALSE

> x < y
[1] TRUE FALSE FALSE
```

Auch die Prüfung jedes Elements auf mehrere Kriterien ist möglich. Wenn zwei Kriterien gleichzeitig erfüllt sein sollen, wird `&` als Symbol für das logische UND verwendet. Wenn nur eines von zwei Kriterien erfüllt sein muss, ist das Symbol `|` für das logische, d. h. einschließende, ODER zu verwenden. Um sicherzustellen, dass R die zusammengehörenden Ausdrücke auch als Einheit erkennt, ist die Verwendung runder Klammern zu empfehlen.

```
> (age <= 20) | (age >= 30)           # Werte im Bereich bis 20 ODER ab 30?
[1] TRUE TRUE TRUE FALSE FALSE FALSE

> (age > 20) & (age < 30)           # Werte im Bereich zwischen 20 und 30?
[1] FALSE FALSE FALSE TRUE TRUE TRUE
```

UND und ODER dürfen bei zusammengesetzten Prüfungen nicht weggelassen werden: Während man mathematisch also eine Bedingung etwa als $0 \leq x \leq 10$ formulieren würde, müsste sie in R in Form von mit UND verbundenen Einzelprüfungen geschrieben werden, also wie oben als $(0 \leq x) \& (x \leq 10)$.

Während die elementweise Prüfung von Vektoren den häufigsten Fall der Anwendung logischer Kriterien ausmacht, sind vor allem zur Fallunterscheidung (vgl. Abschn. 15.1.1) auch Prüfungen notwendig, die in Form eines einzelnen Wahrheitswertes eine summarische Auskunft darüber liefern, ob Kriterien erfüllt sind. Diese Prüfungen lassen sich mit `&&` für das logische UND bzw.

mit `||` für das logische ODER formulieren. Beide Vergleiche werten nur das jeweils erste Element aus, wenn Vektoren beteiligt sind und geben auch in diesem Fall nur einen Wahrheitswert zurück.

```
> c(TRUE, FALSE, FALSE) && c(TRUE, TRUE, FALSE)
[1] TRUE
```

```
> c(FALSE, FALSE, TRUE) || c(FALSE, TRUE, FALSE)
[1] FALSE
```

`identical()` prüft zwei übergebene Vektoren summarisch auf Gleichheit und gibt nur dann das Ergebnis `TRUE` aus, wenn diese auch bzgl. ihrer internen Repräsentation exakt identisch sind (vgl. Abschn. 1.3.6).

```
> c(1, 2) == c(1L, 2L)
[1] TRUE TRUE
```

```
> identical(c(1, 2), c(1L, 2L))
[1] FALSE
```

Sollen Werte nur auf ungefähre Übereinstimmung geprüft werden, kann dies mit `all.equal()` geschehen (vgl. Abschn. 1.3.6). Dabei ist im Fall von zu vergleichenden Vektoren zu beachten, dass die Funktion keinen Vektor der Ergebnisse der elementweisen Einzelvergleiche ausgibt. Stattdessen liefert sie nur einen einzelnen Wert zurück, entweder `TRUE` im Fall der paarweisen Übereinstimmung aller Elemente oder das mittlere Abweichungsmaß im Fall der Ungleichheit. Um auch in letzterem Fall einen Wahrheitswert als Ausgabe zu erhalten, sollte `isTRUE()` verwendet werden.

```
> x <- c(4, 5, 6)
> y <- c(4, 5, 6)
> z <- c(1, 2, 3)
> all.equal(x, y)
[1] TRUE

> all.equal(y, z)
[1] "Mean relative difference: 0.6"

> isTRUE(all.equal(y, z))
[1] FALSE
```

Bei der Prüfung von Elementen auf Kriterien kann mit Hilfe spezialisierter Funktionen summarisch analysiert werden, ob diese Kriterien zutreffen. Ob mindestens ein Element eines logischen Vektors den Wert `TRUE` besitzt, zeigt `any(<Vektor>)`, ob alle Elemente den Wert `TRUE` haben, gibt `all(<Vektor>)` an.⁸

⁸Dabei erzeugt `all(numeric(0))` das Ergebnis `TRUE`, da die Aussage „alle Elemente des leeren Vektors sind WAHR“ logisch WAHR ist – schließlich lässt sich kein Gegenbeispiel in Form eines Elements finden, das FALSCH wäre. Dagegen erzeugt `any(numeric(0))` das Ergebnis `FALSE`, da in einem leeren Vektor nicht mindestens ein Element existiert, das WAHR ist.

```
> res <- age > 30
> any(res)
[1] FALSE

> any(age < 18)
[1] TRUE

> all(x == y)
[1] TRUE
```

Um zu zählen, auf wie viele Elemente eines Vektors ein Kriterium zutrifft, wird die Funktion `sum(<Vektor>)` verwendet, die alle Werte eines Vektors aufaddiert (vgl. Abschn. 2.7.1).

```
> res <- age < 24
> sum(res)
[1] 3
```

Alternativ kann verschachtelt in `length()` die Funktion `which(<Vektor>)` genutzt werden, die die Indizes der Elemente mit dem Wert **TRUE** ausgibt (vgl. Abschn. 2.2.2).

```
> which(age < 24)
[1] 1 5 6

> length(which(age < 24))
[1] 3
```

2.2.2 Logische Indexvektoren

Vektoren von Wahrheitswerten können wie numerische Indexvektoren zur Indizierung anderer Vektoren benutzt werden. Diese Art zu indizieren kann z. B. zur Auswahl von Teilstichproben genutzt werden, die durch bestimmte Merkmale definiert sind. Hat ein Element des logischen Indexvektors den Wert **TRUE**, so wird das sich an dieser Position befindliche Element des indizierten Vektors ausgegeben. Hat der logische Indexvektor an einer Stelle den Wert **FALSE**, so wird das zugehörige Element des indizierten Vektors ausgelassen.

```
> age[c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE)]
[1] 17 30 24 21
```

Wie numerische können auch logische Indizes zunächst in einem Vektor gespeichert werden, mit dem die Indizierung dann später geschieht. Statt der Erstellung eines separaten logischen Indexvektors, z. B. als Ergebnis einer Überprüfung von Bedingungen, kann der Schritt aber auch übersprungen und der logische Ausdruck direkt innerhalb des `[<Index>]` Operators benutzt werden. Dabei ist jedoch abzuwägen, ob der Übersichtlichkeit und Nachvollziehbarkeit der Befehle mit einer separaten Erstellung von Indexvektoren besser gedient ist.

```
> (idx <- (age <= 20) | (age >= 30)) # Werte im Bereich bis 20 ODER ab 30?
[1] TRUE TRUE TRUE FALSE FALSE FALSE
```

```
> age[idx]
[1] 17 30 30

> age[(age >= 30) | (age <= 20)]
[1] 17 30 30
```

Bei logischen Indexvektoren kann anders als bei numerischen Indexvektoren die von R automatisch vorgenommene zyklische Verlängerung greifen (vgl. Abschn. 2.5.4): Logische Indexvektoren mit weniger Elementen als jene des indizierten Vektors werden durch zyklische Wiederholung soweit verlängert, dass sie mindestens die Länge des indizierten Vektors erreichen.

```
> age[c(TRUE, FALSE)]          # logischer Indexvektor kürzer als age
[1] 18 30 23

# durch zyklische Verlängerung von c(TRUE, FALSE) äquivalent zu
> age[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)]
[1] 18 30 23
```

Logische Indexvektoren bergen den Nachteil, dass sie zu Problemen führen können, wenn der zu prüfende Vektor fehlende Werte enthält. Überall dort, wo dieser NA ist, wird i. d. R. auch das Ergebnis eines logischen Vergleichs NA sein, d. h. der resultierende logische Indexvektor enthält seinerseits fehlende Werte (vgl. Abschn. 2.11.3, Fußnote 44).

```
> vecNA    <- c(-3, 2, 0, NA, -7, 5)      # Vektor mit fehlendem Wert
> (logIdx <- vecNA > 0)                  # prüfe auf Werte größer 0
[1] FALSE TRUE FALSE NA FALSE TRUE
```

Enthält ein Indexvektor einen fehlenden Wert, erzeugt er beim Indizieren eines anderen Vektors an dieser Stelle ebenfalls ein NA in der Ausgabe (vgl. Abschn. 2.1.2). Dies führt dazu, dass sich der Indexvektor nicht mehr dazu eignet, ausschließlich die Werte auszugeben, die eine bestimmte Bedingung erfüllen.

```
> vecNA[logIdx]                         # Auswahl mit logIdx erzeugt NA
[1] 2 NA 5
```

In Situationen, in denen fehlende Werte möglich sind, ist deshalb ein anderes Vorgehen sinnvoller: Statt eines logischen Indexvektors sollten die numerischen Indizes derjenigen Elemente zum Indizieren verwendet werden, die die geprüfte Bedingung erfüllen, an deren Position der logische Vektor also den Wert TRUE besitzt. Logische in numerische Indizes wandelt in diesem Sinne die Funktion `which(Vektor)` um, die die Indizes der TRUE Werte zurückgibt.⁹

```
> (numIdx <- which(logIdx))           # numer. Indizes der TRUE Werte
[1] 2 6

> vecNA[numIdx]                      # korrekte Auswahl
[1] 2 5
```

⁹Umgekehrt lassen sich auch die in `\langle Indexvektor \rangle` gespeicherten numerischen Indizes für `\langle Vektor \rangle` in logische verwandeln: `seq(along=\langle Vektor \rangle) %in% \langle Indexvektor \rangle` (vgl. Abschn. 2.3.2, 2.4.1).

2.3 Mengen

Werden Vektoren als Wertemengen im mathematischen Sinn betrachtet, ist zu beachten, dass die Elemente einer Menge nicht geordnet sind und mehrfach vorkommende Elemente wie ein einzelnes behandelt werden, so ist z. B. die Menge $\{1, 1, 2, 2\}$ gleich der Menge $\{2, 1\}$.¹⁰

2.3.1 Doppelt auftretende Werte finden

`duplicated(<Vektor>)` gibt für jedes Element eines Vektors an, ob der Wert bereits an einer früheren Stelle des Vektors aufgetaucht ist. Mit dem Argument `fromLast=TRUE` beginnt die Suche nach wiederholt auftretenden Werten am Ende des Vektors.

```
> x <- c(1, 2, 3, 1, 4, 5)
> duplicated(x)
[1] FALSE FALSE FALSE TRUE FALSE FALSE

> duplicated(x, fromLast=TRUE)           # umgekehrte Suchrichtung
[1] TRUE FALSE FALSE FALSE FALSE FALSE
```

Durch die Kombination beider Suchrichtungen lassen sich alle Duplikate identifizieren, insbesondere also auch das erste Auftreten eines mehrfach vorhandenen Wertes – so umgesetzt in `AllDuplicated()` aus dem Paket `DescTools` (Signorell, 2014).

```
# markiere alle Stellen, an denen die 1 auftritt
> duplicated(x) | duplicated(x, fromLast=TRUE)
[1] TRUE FALSE FALSE TRUE FALSE FALSE
```

`unique(<Vektor>)` nennt alle voneinander verschiedenen Werte eines Vektors, mehrfach vorkommende Werte werden also nur einmal aufgeführt. Die Funktion eignet sich in Kombination mit `length()` zum Zählen der tatsächlich vorkommenden unterschiedlichen Werte einer Variable.

```
> unique(c(1, 1, 1, 3, 3, 4))
[1] 1 3 4

> length(unique(c("A", "B", "C", "C", "B", "B", "A", "C", "C", "A")))
[1] 3
```

2.3.2 Mengenoperationen

`union(x=<Vektor1>, y=<Vektor2>)` bildet die Vereinigungsmenge $x \cup y$. Das Ergebnis sind die Werte, die Element mindestens einer der beiden Mengen sind, wobei duplizierte Werte gelöscht werden. Wird das Ergebnis als Menge betrachtet, spielt es keine Rolle, in welcher Reihenfolge `x` und `y` genannt werden.

¹⁰Das Paket `sets` (Meyer & Hornik, 2009) stellt eine eigene Klasse zur Repräsentation von Mengen zur Verfügung und implementiert auch einige hier nicht behandelte Mengenoperationen – etwa das Bilden der Potenzmenge.

```
> x <- c(2, 1, 3, 2, 1)
> y <- c(5, 3, 1, 3, 4, 4)
> union(x, y)
[1] 2 1 3 5 4

> union(y, x)
[1] 5 3 1 4 2
```

Die Schnittmenge $x \cap y$ zweier Mengen erzeugt `intersect(x=<Vektor1>, y=<Vektor2>)`. Das Ergebnis sind die Werte, die sowohl Element von x als auch Element von y sind, wobei duplizierte Werte gelöscht werden. Auch hier ist die Reihenfolge von x und y unerheblich, wenn das Ergebnis als Menge betrachtet wird.

```
> intersect(x, y)
[1] 1 3

> intersect(y, x)
[1] 3 1
```

Mit `setequal(x=<Vektor1>, y=<Vektor2>)` lässt sich prüfen, ob als Mengen betrachtete Vektoren identisch sind.

```
> setequal(c(1, 1, 2, 2), c(2, 1))
[1] TRUE
```

`setdiff(x=<Vektor1>, y=<Vektor2>)` liefert als Ergebnis all jene Elemente von x , die nicht Element von y sind. Im Unterschied zu den oben behandelten Mengenoperationen ist die Reihenfolge von x und y hier bedeutsam, auch wenn das Ergebnis als Menge betrachtet wird. Die symmetrische Differenz von x und y erhält man durch `union(setdiff(x, y), -setdiff(y, x))`.

```
> setdiff(x, y)
[1] 2

> setdiff(y, x)
[1] 5 4
```

Soll jedes Element eines Vektors daraufhin geprüft werden, ob es Element einer Menge ist, kann `is.element(el=<Vektor>, set=<Menge>)` genutzt werden. Unter `el` ist der Vektor mit den zu prüfenden Elementen einzutragen und unter `set` die durch einen Vektor definierte Menge. Als Ergebnis wird ein logischer Vektor ausgegeben, der für jedes Element von `el` angibt, ob es in `set` enthalten ist. Die Kurzform in Operator-Schreibweise lautet `<Vektor> %in% <Menge>`.

```
> is.element(c(29, 23, 30, 17, 30, 10), c(30, 23))
[1] FALSE TRUE TRUE FALSE TRUE FALSE

> c("A", "Z", "B") %in% c("A", "B", "C", "D", "E")
[1] TRUE FALSE TRUE
```

Durch `all(x %in% y)` lässt sich prüfen, ob `x` eine Teilmenge von `y` darstellt, ob also jedes Element von `x` auch Element von `y` ist. Dabei ist `x` eine echte Teilmenge von `y`, wenn sowohl `all(x %in% y)` gleich TRUE als auch `all(y %in% x)` gleich FALSE ist.

```
> A <- c(4, 5, 6)
> B <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
> (AinB <- all(A %in% B))           # A Teilmenge von B?
[1] TRUE

> (BinA <- all(B %in% A))           # B Teilmenge von A?
[1] FALSE

> AinB & !BinA                      # A echte Teilmenge von B?
[1] TRUE
```

2.3.3 Kombinatorik

Aus dem Bereich der Kombinatorik sind bei der Datenauswertung drei Themen von Bedeutung: Zunächst ist dies die Bildung von Teilmengen aus Elementen einer Grundmenge. Dabei ist die Reihenfolge der Elemente innerhalb einer Teilmenge meist nicht bedeutsam, d. h. es handelt sich um eine *Kombination*. Werden alle Elemente einer Grundmenge ohne Zurücklegen unter Beachtung der Reihenfolge gezogen, handelt es sich um eine vollständige *Permutation*. Schließlich kann die Zusammenstellung von Elementen aus verschiedenen Grundmengen notwendig sein, wobei jeweils ein Element aus jeder Grundmenge beteiligt sein soll.

Die Kombination entspricht dem Ziehen aus einer Grundmenge ohne Zurücklegen sowie ohne Berücksichtigung der Reihenfolge. Oft wird die Anzahl der Elemente der Grundmenge mit n , die Anzahl der gezogenen Elemente mit k und die Kombination deshalb mit k -Kombination bezeichnet. Es gibt $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ viele k -Kombinationen. Da eine k -Kombinationen die Anzahl der Möglichkeiten darstellt, aus einer Menge mit n Elementen k auszuwählen, spricht man im Englischen beim Binomialkoeffizienten $\binom{n}{k}$ von „ n choose k “, woraus sich der Name der `choose(n=⟨Zahl⟩, k=⟨Zahl⟩)` Funktion ableitet, die $\binom{n}{k}$ ermittelt. Die Fakultät einer Zahl wird mit `factorial(⟨Zahl⟩)` berechnet (vgl. Abschn. 2.7.1).

```
> myN <- 5
> myK <- 4
> choose(myN, myK)
[1] 5

> factorial(myN) / (factorial(myK)*factorial(myN-myK))    # Kontrolle
[1] 5
```

Möchte man alle k -Kombinationen einer gegebenen Grundmenge `x` auch explizit anzeigen lassen, kann dies mit `combn()` geschehen.

```
> combn(x=⟨Vektor⟩, m=⟨Zahl⟩, simplify=TRUE, FUN=⟨Funktion⟩, ...)
```

Die Zahl m entspricht dabei dem k der bisherigen Terminologie. Mit `simplify=TRUE` erfolgt die Ausgabe auf möglichst einfache Weise, d. h. nicht als Liste (vgl. Abschn. 3.1). Stattdessen wird hier eine Matrix ausgegeben, die in jeder Spalte eine der k -Kombinationen enthält (vgl. Abschn. 2.8).

```
> combn(c("a", "b", "c", "d", "e"), myK)
     [,1]  [,2]  [,3]  [,4]  [,5]
[1,] "a"   "a"   "a"   "a"   "b"
[2,] "b"   "b"   "b"   "c"   "c"
[3,] "c"   "c"   "d"   "d"   "d"
[4,] "d"   "e"   "e"   "e"   "e"
```

`combn()` lässt sich darüber hinaus anwenden, um in einem Arbeitsschritt eine frei wählbare Funktion auf jede gebildete k -Kombination anzuwenden. Das Argument `FUN` erwartet hierfür eine Funktion, die einen Kennwert jedes sich als Kombination ergebenden Vektors bestimmt. Benötigt `FUN` ihrerseits weitere Argumente, so können diese unter ... durch Komma getrennt an `combn()` übergeben werden.

```
> combn(c(1, 2, 3, 4), 3)           # alle 3-Kombinationen
     [,1]  [,2]  [,3]  [,4]
[1,]    1    1    1    2
[2,]    2    2    3    3
[3,]    3    4    4    4

# jeweilige Summe jeder 3-Kombination
> combn(c(1, 2, 3, 4), 3, FUN=sum)
[1] 6 7 8 9

# gewichtetes Mittel jeder 3-Kombination mit Argument w für Gewichte
> combn(c(1, 2, 3, 4), 3, weighted.mean, w=c(0.5, 0.2, 0.3))
[1] 1.8 2.1 2.3 2.8
```

`Permn(<Vektor>)` aus dem Paket `DescTools` stellt alle $n!$ Permutationen des übergebenen Vektors der Länge n als Zeilen einer Matrix zusammen (vgl. Abschn. 2.8). Für eine einzelne zufällige Permutation vgl. Abschn. 2.4.3.

```
> library(DescTools)               # für Permn()
> Permn(1:3)
     [,1]  [,2]  [,3]
[1,]    1    2    3
[2,]    1    3    2
[3,]    3    1    2
[4,]    3    2    1
[5,]    2    3    1
[6,]    2    1    3
```

`expand.grid(<Vektor1>, <Vektor2>, ...)` bildet das kartesische Produkt der übergebenen Grundmengen. Dies sind alle Kombinationen der Elemente der Grundmengen, wobei jeweils ein Element aus jeder Grundmenge stammt und die Reihenfolge nicht berücksichtigt wird.

Dies entspricht der Situation, dass aus den Stufen mehrerer Faktoren alle Kombinationen von Faktorstufen gebildet werden. Das Ergebnis von `expand.grid()` ist ein Datensatz (vgl. Abschn. 3.2), bei dem jede Kombination in einer Zeile steht. Die zuerst genannte Variable variiert dabei am schnellsten über die Zeilen, die anderen entsprechend ihrer Position im Funktionsaufruf langsamer.

```
> IV1 <- c("control", "treatment")
> IV2 <- c("f", "m")
> IV3 <- c(1, 2)
> expand.grid(IV1, IV2, IV3)
   Var1  Var2  Var3
1 control    f    1
2 treatment  f    1
3 control    m    1
4 treatment  m    1
5 control    f    2
6 treatment  f    2
7 control    m    2
8 treatment  m    2
```

Variablen mit Zeichenketten werden von `expand.grid()` automatisch in Gruppierungsfaktoren (Klasse `factor`) konvertiert. Sollen solche Variablen als `character` Vektoren gespeichert werden, ist das Argument `stringsAsFactors=FALSE` zu setzen. Sollen auch numerische Werte zu Faktorstufen im versuchsplanerischen Sinn werden (was hier für die dritte Spalte nicht der Fall ist), muss die zugehörige Variable vorher in ein Objekt der Klasse `factor` umgewandelt werden.

2.4 Systematische und zufällige Wertefolgen erzeugen

Ein häufig auftretender Arbeitsschritt in R ist die Erstellung von Zahlenfolgen nach vorgegebenen Gesetzmäßigkeiten, wie etwa sequentielle Abfolgen von Zahlen oder Wiederholungen bestimmter Wertemuster.

Aber auch Zufallszahlen und zufällige Reihenfolgen sind ein unverzichtbares Hilfsmittel der Datenauswertung, wobei ihnen insbesondere in der Planung von Analysen anhand simulierter Daten eine große Bedeutung zukommt.¹¹ Zufällige Datensätze können unter Einhaltung vorgegebener Wertebereiche und anderer Randbedingungen erstellt werden. So können sie empirische Gegebenheiten realistisch widerspiegeln und statistische Voraussetzungen der eingesetzten

¹¹Wenn hier und im folgenden von Zufallszahlen die Rede ist, sind immer *Pseudozufallszahlen* gemeint. Diese kommen nicht im eigentlichen Sinn zufällig zustande, sind aber von tatsächlich zufälligen Zahlenfolgen fast nicht zu unterscheiden. Pseudozufallszahlen hängen deterministisch vom Zustand des die Zahlen produzierenden Generators ab. Wird sein Zustand über `set.seed(<Zahl>)` festgelegt, kommt bei gleicher `(Zahl)` bei späteren Aufrufen von Zufallsfunktionen immer dieselbe Folge von Werten zustande. Dies gewährleistet die Reproduzierbarkeit von Auswertungsschritten bei Simulationen. Nach welcher Methode Zufallszahlen generiert werden, ist konfigurierbar, vgl. `?RNGkind`.

Verfahren berücksichtigen. Aber auch bei der zufälligen Auswahl von Teilstichproben eines Datensatzes oder beim Erstellen zufälliger Reihenfolgen zur Zuordnung von Beobachtungsobjekten auf experimentelle Bedingungen kommen Zufallszahlen zum Einsatz.

2.4.1 Numerische Sequenzen erstellen

Zahlenfolgen mit Einerschritten, etwa für eine fortlaufende Numerierung, können mit Hilfe des Operators `<Startwert>:<Endwert>` erzeugt werden – in aufsteigender wie auch in absteigender Reihenfolge.

```
> 20:26
[1] 20 21 22 23 24 25 26

> 26:20
[1] 26 25 24 23 22 21 20
```

Bei Zahlenfolgen im negativen Bereich sollten Klammern Verwendung finden, um nicht versehentlich eine nicht gemeinte Sequenz zu produzieren.

```
> -4:-2          # negatives Vorzeichen bezieht sich nur auf die 4
[1] -4 -3 -2 -1 0 1 2

> -(4:2)         # negatives Vorzeichen bezieht sich auf Sequenz 4:2
[1] -4 -3 -2
```

Zahlenfolgen mit beliebiger Schrittweite lassen sich mit `seq()` erzeugen.

```
> seq(from=<Zahl>, to=<Zahl>, by=<Schrittweite>, length.out=<Länge>)
```

Dabei können Start- (`from`) und Endwert (`to`) des durch die Sequenz abzudeckenden Intervalls ebenso gewählt werden wie die gewünschte Schrittweite (`by`) bzw. stattdessen die gewünschte Anzahl der Elemente der Zahlenfolge (`length.out`). Die Sequenz endet vor `to`, wenn die Schrittweite kein ganzzahliges Vielfaches der Differenz von Start- und Endwert ist.

```
> seq(from=2, to=12, by=2)
[1] 2 4 6 8 10 12

> seq(from=2, to=11, by=2)           # Endpunkt 11 wird nicht erreicht
[1] 2 4 6 8 10

> seq(from=0, to=-1, length.out=5)
[1] 0.00 -0.25 -0.50 -0.75 -1.00
```

Eine Möglichkeit zum Erstellen einer bei 1 beginnenden Sequenz in Einerschritten, die genauso lang ist wie ein bereits vorhandener Vektor, besteht mit `seq(along=<Vektor>)`. Dabei muss `along=<Vektor>` das einzige Argument von `seq()` sein. Dies ist die bevorzugte Art, um für einen vorhandenen Vektor den passenden Vektor seiner Indizes zu erstellen. Vermieden werden sollte dagegen die `1:length(<Vektor>)` Sequenz, deren Behandlung von Vektoren der Länge 0 nicht sinnvoll ist.

```
> age <- c(18, 20, 30, 24, 23, 21)
> seq(along=age)
[1] 1 2 3 4 5 6

> vec <- numeric(0)          # leeren Vektor (Länge 0) erzeugen
> 1:length(vec)             # hier unerwünschtes Ergebnis: Sequenz 1:0
[1] 1 0

> seq(along=vec)            # sinnvolleres Ergebnis: leerer Vektor
[1] integer(0)
```

2.4.2 Wertefolgen wiederholen

Eine andere Art von Wertefolgen kann mit der `rep()` Funktion (repeat) erzeugt werden, die Elemente wiederholt ausgibt.

```
> rep(x=<Vektor>, times=<Anzahl>, each=<Anzahl>)
```

Für `x` ist ein Vektor einzutragen, der auf zwei verschiedene Arten wiederholt werden kann. Mit dem Argument `times` wird er als Ganzes so oft aneinander gehängt wie angegeben.

```
> rep(1:3, times=5)
[1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

Wird für das Argument `times` ein Vektor angegeben, so muss dieser dieselbe Länge wie `x` besitzen – hier wird ein kürzerer Vektor durch R nicht selbsttätig zyklisch wiederholt (vgl. Abschn. 2.5.4). Jedes Element des Vektors `times` gibt an, wie häufig das an gleicher Position stehende Element von `x` wiederholt werden soll, ehe das nächste Element von `x` wiederholt und angehängt wird.

```
> rep(c("A", "B", "C"), times=c(2, 3, 4))
[1] "A" "A" "B" "B" "B" "C" "C" "C" "C"
```

Wird das Argument `each` verwendet, wird jedes Element von `x` einzeln mit der gewünschten Häufigkeit wiederholt, bevor das nächste Element von `x` einzeln wiederholt und angehängt wird.

```
> rep(age, each=2)
[1] 18 18 20 20 30 30 24 24 23 23 21 21
```

2.4.3 Zufällig aus einer Urne ziehen

Die Funktion `sample()` erstellt einen aus zufälligen Werten bestehenden Vektor, indem sie das Ziehen aus einer Urne simuliert.

```
> sample(x=<Vektor>, size=<Anzahl>, replace=FALSE, prob=NULL)
```

Für `x` ist ein Vektor zu nennen, der die Elemente der Urne festlegt, aus der gezogen wird. Dies sind die Werte, aus denen sich die Zufallsfolge zusammensetzen soll. Es können Vektoren vom Datentyp `numeric` (etwa `1:50`), `character` (`c("Kopf", "Zahl")`) oder auch `logical` (`c(TRUE, FALSE)`) verwendet werden. Unter `size` ist die gewünschte Anzahl der zu ziehenden Elemente einzutragen. Mit dem Argument `replace` wird die Art des Ziehens festgelegt: Auf `FALSE` gesetzt (Voreinstellung) wird ohne, andernfalls (`TRUE`) mit Zurücklegen gezogen. Natürlich kann ohne Zurücklegen aus einem Vektor der Länge n nicht häufiger als n mal gezogen werden. Wenn `replace=FALSE` und dennoch `size` größer als `length(x)` ist, erzeugt R deswegen eine Fehlermeldung. Für den Fall, dass nicht alle Elemente der Urne dieselbe Auftretenswahrscheinlichkeit besitzen sollen, existiert das Argument `prob`. Es benötigt einen Vektor derselben Länge wie `x`, dessen Elemente die Auftretenswahrscheinlichkeit für jedes Element von `x` bestimmen.

```
> sample(1:6, size=20, replace=TRUE)
[1] 4 1 2 5 6 5 3 6 6 5 1 6 1 5 1 4 5 4 4 2

> sample(c("rot", "grün", "blau"), size=8, replace=TRUE)
[1] "grün" "blau" "grün" "rot" "rot" "blau" "grün" "blau"
```

Für `sample()` existieren zwei Kurzformen, auf die jedoch aufgrund der Gefahr von Verwechslungen besser verzichtet werden sollte: `sample(<Vektor>)` ist gleichbedeutend mit `sample(<Vektor>, size=length(<Vektor>), replace=FALSE)`, erstellt also eine zufällige Permutation der Elemente von `<Vektor>` (vgl. Abschn. 2.3.3). Darauf aufbauend steht `sample(<Zahl>)` kurz für `sample(1:<Zahl>)`, also für `sample(1:<Zahl>, size=<Zahl>, replace=FALSE)`.

Wenn für `sample(<Vektor>)` ein Objektname übergeben wird, steht oft vor der Ausführung nicht fest, wie viele Elemente er beinhaltet wird. Enthält `<Vektor>` z. B. durch die Auswahl einer Teilmenge unvorhergesehen als einziges Element eine Zahl, wird die Urne durch Elemente definiert, die womöglich nicht im ursprünglichen Vektor vorhanden waren.

```
> x <- c(2, 4, 6, 8)
> sample(x[(x %% 4) == 0])      # äquivalent zu sample(c(4, 8))
[1] 8 4

# Urne mit Elementen, die nicht aus x stammen
> sample(x[(x %% 8) == 0])      # äquivalent zu sample(8), d.h. sample(1:8)
[1] 2 1 7 5 4 8 6 3
```

2.4.4 Zufallszahlen aus bestimmten Verteilungen erzeugen

Abgesehen vom zufälligen Ziehen aus einer vorgegebenen Menge endlich vieler Werte lassen sich auch Zufallszahlen mit bestimmten Eigenschaften generieren. Dazu können mit Funktionen, deren Name nach dem Muster `r(Funktionsfamilie)` aufgebaut ist, Realisierungen von Zufallsvariablen mit verschiedenen Verteilungen erstellt werden (vgl. Abschn. 5.3). Diese Möglichkeit ist insbesondere für die Simulation empirischer Daten nützlich.

```
> runif(n=<Anzahl>, min=0, max=1)                      # Gleichverteilung
> rbinom(n=<Anzahl>, size, prob)                         # Binomialverteilung
> rnorm(n=<Anzahl>, mean=0, sd=1)                         # Normalverteilung
```

```
> rchisq(n=(Anzahl), df, ncp=0) # chi^2-Verteilung
> rt(n=(Anzahl), df, ncp=0) # t-Verteilung
> rf(n=(Anzahl), df1, df2, ncp=0) # F-Verteilung
```

Als erstes Argument **n** ist immer die gewünschte Anzahl an Zufallszahlen anzugeben. Bei **runif()** definiert **min** die untere und **max** die obere Grenze des Zahlenbereichs, aus dem gezogen wird. Beide Argumente akzeptieren auch Vektoren der Länge *n*, die für jede einzelne Zufallszahl den zulässigen Wertebereich angeben.

Bei **rbinom()** entsteht jede der *n* Zufallszahlen als Anzahl der Treffer in einer simulierten Serie von gleichen Bernoulli-Experimenten, die ihrerseits durch die Argumente **size** und **prob** charakterisiert ist. **size** gibt an, wie häufig ein einzelnes Bernoulli-Experiment wiederholt werden soll, **prob** legt die Trefferwahrscheinlichkeit in jedem dieser Experimente fest. Sowohl **size** als auch **prob** können Vektoren der Länge *n* sein, die dann die Bernoulli-Experimente charakterisieren, deren Simulation zu jeweils einer Zufallszahl führt.

Bei **rnorm()** sind der Erwartungswert **mean** und die theoretische Streuung **sd** der normalverteilten Variable anzugeben, die simuliert werden soll.¹² Auch diese Argumente können Vektoren der Länge *n* sein und für jede Zufallszahl andere Parameter vorgeben.

Sind Verteilungen über Freiheitsgrade und Nonzentralitätsparameter charakterisiert, werden diese mit den Argumenten **df** (degrees of freedom) respektive **ncp** (non-centrality parameter) ggf. in Form von Vektoren übergeben.

```
> runif(5, min=1, max=6)
[1] 4.411716 3.893652 2.412720 5.676668 2.446302

> rbinom(20, size=5, prob=0.3)
[1] 2 0 3 0 2 2 1 0 1 0 2 1 1 4 2 2 1 1 3 3

> rnorm(6, mean=100, sd=15)
[1] 101.13170 102.25592 88.31622 101.22469 93.12013 100.52888
```

2.5 Daten transformieren

Häufig sind für spätere Auswertungen neue Variablen auf Basis der erhobenen Daten zu bilden. Im Rahmen solcher Datentransformationen können etwa Werte sortiert, umskaliert, ersetzt, ausgewählt, oder verschiedene Variablen zu einer neuen verrechnet werden. Genauso ist es möglich, kontinuierliche Variablen in Kategorien einzuteilen, oder in Rangwerte umzuwandeln.

2.5.1 Werte sortieren

Um die Reihenfolge eines Vektors umzukehren, kann **rev(<Vektor>)** (reverse) benutzt werden.

¹²Der die Breite (Dispersion) einer Normalverteilung charakterisierende Parameter ist hier die Streuung σ , in der Literatur dagegen häufig die Varianz σ^2 .

```
> vec <- c(10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)
> rev(vec)
[1] 20 19 18 17 16 15 14 13 12 11 10
```

Die Elemente eines Vektors lassen sich auch entsprechend ihrer Reihenfolge sortieren, die wiederum vom Datentyp des Vektors abhängt: Bei numerischen Vektoren bestimmt die Größe der gespeicherten Zahlen, bei Vektoren aus Zeichenketten die alphabetische Reihenfolge der Elemente die Ausgabe.¹³ Zum Sortieren stehen die Funktionen `sort()` und `order()` zur Verfügung.

```
> sort(<Vektor>, decreasing=FALSE)
> order(<Vektor>, decreasing=FALSE)
```

`sort()` gibt den sortierten Vektor direkt aus. Dagegen ist das Ergebnis von `order()` ein Indexvektor, der die Indizes des zu ordnenden Vektors in der Reihenfolge seiner Elemente enthält. Im Gegensatz zu `sort()` gibt `order()` also nicht schon die sortierten Datenwerte, sondern nur die zugehörigen Indizes aus, die anschließend zum Indizieren des Vektors verwendet werden können. Für einen Vektor `x` ist daher `sort(x)` äquivalent zu `x[order(x)]`. Der Vorteil von `order()` erweist sich beim Umgang mit Matrizen und Datensätzen (vgl. Abschn. 2.8.6). Die Sortierreihenfolge wird über das Argument `decreasing` kontrolliert. In der Voreinstellung `FALSE` wird aufsteigend sortiert. Mit `decreasing=TRUE` ist die Reihenfolge absteigend.

```
> vec <- c(10, 12, 1, 12, 7, 16, 6, 19, 10, 19)
> sort(vec)
[1] 1 6 7 10 10 12 12 16 19 19

> (idxDec <- order(vec, decreasing=TRUE))
[1] 8 10 6 2 4 1 9 5 7 3

> vec[idxDec]
[1] 19 19 16 12 12 10 10 7 6 1
```

Wenn Vektoren vom Datentyp `character` sortiert werden, so geschieht dies in alphabetischer Reihenfolge. Auch als Zeichenkette gespeicherte Zahlen werden hierbei alphabetisch sortiert, d. h. die Zeichenkette "10" käme vor "4".

```
> sort(c("D", "E", "10", "A", "F", "E", "D", "4", "E", "A"))
[1] "10" "4" "A" "A" "D" "D" "E" "E" "E" "F"
```

2.5.2 Werte in zufällige Reihenfolge bringen

Zufällige Reihenfolgen können mit Kombinationen von `rep()` und `sample()` erstellt werden und entsprechen der zufälligen Permutation einer Menge (vgl. Abschn. 2.3.3). Sie sind z. B. bei der randomisierten Zuteilung von Beobachtungsobjekten zu Gruppen, beim Randomisieren der Reihenfolge von Bedingungen oder beim Ziehen einer Zufallsstichprobe aus einer Datenmenge nützlich.

¹³Die alphabetische Reihenfolge hängt dabei von den Ländereinstellungen ab, die sich mit `Sys.getlocale()` erfragen und mit `Sys.setlocale()` ändern lässt.

```
# randomisiere Reihenfolge von 5 Farben
> myColors <- c("red", "green", "blue", "yellow", "black")
> (randCols <- sample(myColors, length(myColors), replace=FALSE))
[1] "yellow" "green" "red" "blue" "black"

# teile 12 Personen auf 3 unterschiedlich große Gruppen auf
> P <- 3 # Anzahl Gruppen
> Nj <- c(4, 3, 5) # Gruppengrößen
> (IV <- rep(1:P, Nj)) # Gruppenzugehörigkeiten
[1] 1 1 1 1 2 2 2 3 3 3 3 3

# zufällige Permutation
> (IVrand <- sample(IV, length(IV), replace=FALSE))
[1] 2 1 1 3 3 1 3 1 2 2 3 3
```

Um allgemein n Beobachtungsobjekte auf p möglichst ähnlich große Gruppen aufzuteilen, können zunächst mit `sample()` die Indizes $1, \dots, n$ permutiert werden, um dann mit `%%` den Rest der ganzzahligen Division jedes Index mit p zu bilden, der die p Werte $0, \dots, p-1$ annehmen kann.

```
> P <- 3
> N <- 20
> (sample(1:N, N, replace=FALSE) %% P) + 1
[1] 2 2 2 3 3 3 1 3 1 2 3 2 1 1 2 3 2 1 3 1
```

2.5.3 Teilmengen von Daten auswählen

Soll aus einer vorhandenen Datenmenge eine Teilstichprobe gezogen werden, hängt das Vorgehen von der genau intendierten Art der Ziehung ab. Grundsätzlich können zur Auswahl von Werten logische wie numerische Indexvektoren zum Einsatz kommen, die sich systematisch oder zufällig erzeugen lassen.

Eine rein zufällige Unterauswahl eines bestimmten Umfangs ohne weitere Nebenbedingungen kann mit `sample()` erstellt werden.¹⁴ Dazu betrachtet man den Datenvektor als Urne, aus der ohne Zurücklegen die gewünschte Anzahl von Beobachtungen gezogen wird.

```
> vec <- rep(c("rot", "grün", "blau"), 10)
> sample(vec, 5, replace=FALSE)
[1] "blau" "grün" "blau" "grün" "rot"
```

Ein anderes Ziel könnte darin bestehen, z. B. jedes zehnte Element einer Datenreihe auszugeben. Hier bietet sich `seq()` an, um die passenden Indizes zu erzeugen.

```
> selIdx1 <- seq(1, length(vec), by=10)
> vec[selIdx1]
[1] "rot"  "grün" "blau"
```

¹⁴Das Paket `car` bietet hierfür die Funktion `some()`, die sich auch für Matrizen (vgl. Abschn. 2.8) oder Datensätze (vgl. Abschn. 3.2) eignet.

Soll nicht genau, sondern nur im Mittel jedes zehnte Element ausgegeben werden, eignet sich `rbinom()` zum Erstellen eines geeigneten Indexvektors. Dazu kann der Vektor der Trefferanzahlen aus einer Serie von jeweils nur einmal durchgeföhrten Bernoulli-Experimenten mit Trefferwahrscheinlichkeit $\frac{1}{10}$ in einen logischen Indexvektor umgewandelt werden:

```
> selIdx2 <- rbinom(length(vec), size=1, prob=0.1) == 1
> vec[selIdx2]
[1] "blau" "grün" "blau" "grün" "grün"
```

2.5.4 Daten umrechnen

Auf Vektoren lassen sich alle elementaren Rechenoperationen anwenden, die in Abschn. 1.2.4 für Skalare aufgeführt wurden. Vektoren können also in den meisten Rechnungen wie Einzelwerte verwendet werden, wodurch sich Variablen leicht umskalieren lassen. Die Berechnungen einer Funktion werden dafür elementweise durchgeföhr: Die Funktion wird zunächst auf das erste Element des Vektors angewendet, dann auf das zweite, usw., bis zum letzten Element. Das Ergebnis ist ein Vektor, der als Elemente die Einzelergebnisse besitzt. In der Konsequenz ähnelt die Schreibweise zur Transformation von in Vektoren gespeicherten Werten in R sehr der aus mathematischen Formeln gewohnten.

```
> age <- c(18, 20, 30, 24, 23, 21)
> age/10
[1] 1.8 2.0 3.0 2.4 2.3 2.1

> (age/2) + 5
[1] 14.0 15.0 20.0 17.0 16.5 15.5
```

Auch in Situationen, in denen mehrere Vektoren in einer Rechnung auftauchen, können diese wie Einzelwerte verwendet werden. Die Vektoren werden dann elementweise entsprechend der gewählten Rechenoperation miteinander verrechnet. Dabei wird das erste Element des ersten Vektors mit dem ersten Element des zweiten Vektors z. B. multipliziert, ebenso das zweite Element des ersten Vektors mit dem zweiten Element des zweiten Vektors, usw.

```
> vec1 <- c(3, 4, 5, 6)
> vec2 <- c(-2, 2, -2, 2)
> vec1*vec2
[1] -6 8 -10 12

> vec3 <- c(10, 100, 1000, 10000)
> (vec1 + vec2) / vec3
[1] 1e-01 6e-02 3e-03 8e-04
```

Die Zahlen der letzten Ausgabe sind in verkürzter Exponentialschreibweise dargestellt (vgl. Abschn. 1.2.4).

Zyklische Verlängerung von Vektoren (recycling)

Die Verrechnung mehrerer Vektoren scheint aufgrund der elementweisen Zuordnung zunächst vorauszusetzen, dass die Vektoren dieselbe Länge haben. Tatsächlich ist dies nicht unbedingt notwendig, weil R in den meisten Fällen diesen Zustand ggf. selbsttätig herstellt. Dabei wird der kürzere Vektor intern von R zyklisch wiederholt (also sich selbst angefügt, sog. *recycling*), bis er mindestens die Länge des längeren Vektors besitzt. Eine Warnmeldung wird in einem solchen Fall nur dann ausgegeben, wenn die Länge des längeren Vektors kein ganzzahliges Vielfaches der Länge des kürzeren Vektors ist. Dies ist gefährlich, weil meist Vektoren gleicher Länge miteinander verrechnet werden sollen und die Verwendung von Vektoren ungleicher Länge ein Hinweis auf fehlerhafte Berechnungen sein kann.

```
> vec1 <- c(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24)
> vec2 <- c(2, 4, 6, 8, 10)

> c(length(age), length(vec1), length(vec2))
[1] 6 12 5

> vec1*age
[1] 36 80 180 192 230 252 252 320 540 480 506 504

> vec2*age
[1] 36 80 180 192 230 42
```

Warning message:

Länge des längeren Objektes ist kein Vielfaches der Länge des kürzeren Objektes in: vec2 * age

z-Transformation

Durch eine *z*-Transformation wird eine quantitative Variable X so normiert, dass sie den Mittelwert $\bar{x} = 0$ und die Standardabweichung $s = 1$ besitzt. Dies geschieht für jeden Einzelwert x_i durch $\frac{x_i - \bar{x}}{s}$. `mean(Vektor)` berechnet den Mittelwert (vgl. Abschn. 2.7.3), `sd(Vektor)` ermittelt die korrigierte Streuung (vgl. Abschn. 2.7.6).

```
> (zAge <- (age - mean(age)) / sd(age))
[1] -1.1166106 -0.6380632 1.7546739 0.3190316 0.0797579 -0.3987895
```

Eine andere Möglichkeit bietet die Funktion `scale(x=Vektor, center=TRUE, scale=TRUE)`. Sie berechnet die *z*-Werte mit Hilfe der korrigierten Streuung, gibt sie jedoch nicht in Form eines Vektors, sondern als Matrix mit einer Spalte aus.¹⁵ Weiterhin werden Mittelwert und korrigierte Streuung von x in Form von Attributen mit angegeben. Standardisierung und Zentrierung können unabhängig voneinander ausgewählt werden: Für die zentrierten, nicht aber standardisierten Werte von x ist etwa `scale=FALSE` zu setzen und `center=TRUE` zu belassen.

¹⁵Für x kann auch eine Matrix übergeben werden, deren *z*-transformierte Spalten dann die Spalten der ausgegebenen Matrix ausmachen (vgl. Abschn. 2.8).

```
> (zAge <- scale(age))
[1,] [,1]
[1,] -1.1166106
[2,] -0.6380632
[3,] 1.7546739
[4,] 0.3190316
[5,] 0.0797579
[6,] -0.3987895

attr("scaled:center")
[1] 22.66667

attr("scaled:scale")
[1] 4.179314
```

Um die ausgegebene Matrix wieder in einen Vektor zu verwandeln, muss sie wie in Abschn. 2.8.2 dargestellt mit `as.vector(<Matrix>)` konvertiert werden.

```
> as.vector(zAge)
[1] -1.1166106 -0.6380632 1.7546739 0.3190316 0.0797579 -0.3987895
```

Durch Umkehrung des Prinzips der z -Transformation lassen sich empirische Datenreihen so skalieren, dass sie einen beliebigen Mittelwert \bar{x}_{neu} und eine beliebige Streuung s_{neu} besitzen. Dies geschieht für eine z -transformierte Variable Z mit $Z \cdot s_{\text{neu}} + \bar{x}_{\text{neu}}$.

```
> newSd   <- 15
> newMean <- 100
> (newAge <- as.vector(zAge)*newSd + newMean)
[1] 83.25084 90.42905 126.32011 104.78547 101.19637 94.01816

> mean(newAge)                                # Kontrolle: Mittelwert
[1] 100

> sd(newAge)                                  # Kontrolle: Streuung
[1] 15
```

Rangtransformation

`rank(<Vektor>)` gibt für jedes Element eines Vektors seinen Rangplatz an, der sich an der Position des Wertes im sortierten Vektor orientiert und damit der Ausgabe von `order()` ähnelt. Anders als bei `order()` erhalten identische Werte in der Voreinstellung jedoch denselben Rang. Das Verhalten, mit dem bei solchen *Bindungen* Ränge ermittelt werden, kontrolliert das Argument `ties.method` – Voreinstellung sind mittlere Ränge.

```
> rank(c(3, 1, 2, 3))
[1] 3.5 1.0 2.0 3.5
```

2.5.5 Neue aus bestehenden Variablen bilden

Das elementweise Verrechnen mehrerer Vektoren kann, analog zur z -Transformation, allgemein zur flexiblen Neubildung von Variablen aus bereits bestehenden Daten genutzt werden.

Ein Beispiel sei die Berechnung des Body-Mass-Index (BMI) einer Person, für den ihr Körpergewicht in kg durch das Quadrat ihrer Körpergröße in m geteilt wird.

```
> height <- c(1.78, 1.91, 1.89, 1.83, 1.64)
> weight <- c(65, 89, 91, 75, 73)
> (bmi <- weight / (height^2))
[1] 20.51509 24.39626 25.47521 22.39541 27.14158
```

In einem zweiten Beispiel soll die Summenvariable aus drei dichotomen Items („trifft zu“: TRUE, „trifft nicht zu“: FALSE) eines an 8 Personen erhobenen Fragebogens gebildet werden. Dies ist die Variable, die jeder Person den Summenscore aus ihren Antworten zuordnet, also angibt, wie viele Items von der Person als zutreffend angekreuzt wurden. Logische Werte verhalten sich bei numerischen Rechnungen wie 1 (TRUE) bzw. 0 (FALSE).

```
> quest1 <- c(FALSE, FALSE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE)
> quest2 <- c(TRUE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, FALSE)
> quest3 <- c(TRUE, TRUE, TRUE, TRUE, FALSE, TRUE, FALSE, FALSE)
> (sumVar <- quest1 + quest2 + quest3)
[1] 2 1 1 2 1 3 1 1
```

2.5.6 Werte ersetzen oder recodieren

Mitunter werden Variablen zunächst auf eine bestimmte Art codiert, die sich später für manche Auswertungen als nicht zweckmäßig erweist und deswegen geändert werden soll. Dann müssen bestimmte Werte gesucht und ersetzt werden, was sich auf verschiedenen Wegen erreichen lässt. Dabei sollte die Variable mit recodierten Werten stets als neues Objekt erstellt werden, statt die Werte des alten Objekts zu überschreiben.

Logische Indexvektoren bieten eine von mehreren Methoden, um systematisch bestimmte Werte durch andere zu ersetzen. In einem Vektor seien dazu die Lieblingsfarben von sieben englischsprachigen Personen erhoben worden. Später soll die Variable auf deutsche Farbnamen recodiert werden.

```
> myColors <- c("red", "purple", "blue", "blue", "orange", "red", "orange")
> farben <- character(length(myColors)) # neuen Vektor erstellen
> farben[myColors == "red"] <- "rot"
> farben[myColors == "purple"] <- "violett"
> farben[myColors == "blue"] <- "blau"
> farben[myColors == "orange"] <- "orange"
> farben
[1] "rot" "violett" "blau" "blau" "orange" "rot" "orange"
```

Mit `replace()` können Werte eines Vektors ebenfalls über Indexvektoren ausgetauscht werden.

```
> replace(x=<Vektor>, list=<Indexvektor>, values=<neue Werte>)
```

Der Vektor mit den auszutauschenden Elementen ist unter `x` zu nennen. Welche Werte geändert werden sollen, gibt der Indexvektor `list` über numerische oder logische Indizes an. Der Vektor `values` definiert für jeden in `list` genannten Index mit je einem Element, welcher Wert an dieser Stelle neu einzufügen ist. Da `replace()` den unter `x` angegebenen Vektor nicht verändert, muss das Ergebnis ggf. einem neuen Objekt zugewiesen werden.

```
> replace(c(1, 2, 3, 4, 5), list=c(2, 4), values=c(200, 400))
[1] 1 200 3 400 5
```

`recode()` aus dem Paket `car` ermöglicht es auf bequemere Weise, gleichzeitig viele Werte nach einem Muster zu ändern, ohne dabei selbst logische Indexvektoren bilden zu müssen.

```
> recode(var=<Vektor>, recodes=<Muster>")
```

Der Vektor mit zu ändernden Werten ist für `var` anzugeben. Die Recodierung erfolgt anhand eines Musters, das das Argument `recodes` bestimmt. Hierbei handelt es sich um eine besonders aufgebaute Zeichenkette: Sie besteht aus durch Semikolon getrennten Elementen, von denen jedes eine Zuordnung von alten und neuen Werten in der Form `<alt>=<neu>` definiert. `<alt>` nennt in Form eines Vektors Werte in `var`, die durch denselben neuen Wert zu ersetzen sind. Sie müssen in `var` nicht unbedingt auch tatsächlich vorkommen, wodurch sich auch potentielle Wertebereiche austauschen lassen. `<neu>` ist der gemeinsame neue Wert für die unter `<alt>` genannten.

Bei alten und neuen Werten sind Zeichenketten jeweils in einfache Anführungszeichen '`<Zeichen>`' zu setzen, wenn `recodes` insgesamt in doppelten Anführungszeichen "`<Zeichen>"` steht. Statt einem konkreten alten Wert kann auch dem Schlüsselwort `else` ein neuer zugewiesen werden, der dann für alle nicht anderweitig umcodierten Werte gilt. Tauchen Werte von `var` nicht im Muster `recodes` auf, bleiben sie unverändert. Auch `recode()` verändert den Vektor mit auszutauschenden Werten selbst nicht, weshalb das Ergebnis ggf. einem neuen Objekt zugewiesen werden muss.

```
> library(car) # für recode()
> recode(myColors, "'red'='rot'; 'blue'='blau'; 'purple'='violett')")
[1] "rot" "violett" "blau" "blau" "orange" "rot" "orange"
```

```
# Einteilung der Farben in Basisfarben und andere
> recode(myColors, "c('red', 'blue')='basic'; else='complex'")
[1] "basic" "complex" "basic" "basic" "complex" "basic" "complex"
```

Gilt es, Werte entsprechend einer dichotomen Entscheidung durch andere zu ersetzen, kann dies mit `ifelse()` geschehen.

```
> ifelse(test=<logischer Ausdruck>, yes=<Wert>, no=<Wert>)
```

Für das Argument `test` muss ein Ausdruck angegeben werden, der sich zu einem logischen Wert auswerten lässt, der also WAHR (`TRUE`) oder FALSCH (`FALSE`) ist. Ist `test` WAHR, wird der unter `yes` eingetragene Wert zurückgegeben, andernfalls der unter `no` genannte. Ist `test` ein Vektor, wird jedes seiner Elemente daraufhin geprüft, ob es `TRUE` oder `FALSE` ist und ein

Vektor mit den passenden, unter **yes** und **no** genannten Werten als Elementen zurückgegeben. Die Ausgabe hat also immer dieselbe Länge wie die von **test**.

Die Argumente **yes** und **no** können selbst Vektoren derselben Länge wie **test** sein – ist dann etwa das dritte Element von **test** gleich TRUE, wird als drittes Element des Ergebnisses das dritte Element von **yes** zurückgegeben, andernfalls das dritte Element von **no**. Indem für **yes** ebenfalls der in **test** zu prüfende Vektor eingesetzt wird, können so bestimmte Werte eines Vektors ausgetauscht, andere dagegen unverändert gelassen werden. Dies erlaubt es etwa, alle Werte größer einem Cutoff-Wert auf denselben Maximalwert zu setzen und die übrigen Werte beizubehalten.

```
> orgVec <- c(5, 9, 11, 8, 9, 3, 1, 13, 9, 12, 5, 12, 6, 3, 17, 5, 8, 7)
> cutoff <- 10
> (reVec <- ifelse(orgVec <= cutoff, orgVec, cutoff))
[1] 5 9 10 8 9 3 1 10 9 10 5 10 6 3 10 5 8 7
```

In Kombination mit **%in%** kann **ifelse()** auch genutzt werden, um eine kategoriale Variable so umzucodieren, dass alle nicht in einer bestimmten Menge auftauchenden Werte in einer Kategorie „Sonstiges“ zusammengefasst werden. Dabei sollte die Variable ein einfacher Vektor sein, da **ifelse()** die Klasse des übergebenen Objekts nicht respektiert – insbesondere Faktoren (vgl. Abschn. 2.6) sind deshalb für **ifelse()** ungeeignet. Als Beispiel sollen nur die ersten 15 Buchstaben des Alphabets als solche beibehalten, alle anderen Werte zu "other" recodiert werden.

```
> targetSet <- c("A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K")
> response <- c("Z", "E", "O", "W", "H", "C", "I", "G", "A", "O", "B")
> (respRec <- ifelse(response %in% targetSet, response, "other"))
[1] "other" "E" "other" "other" "H" "C" "I" "G" "A" "other" "B"
```

2.5.7 Kontinuierliche Variablen in Kategorien einteilen

Als Spezialfall des Recodierens können neue Variablen dadurch entstehen, dass der Wertebereich von ursprünglich kontinuierlichen Variablen in Klassen eingeteilt wird. Auf diese Weise lässt sich eine quantitative in eine kategoriale Variable umwandeln. Hier soll der IQ-Wert mehrerer Personen zu einer Klasseneinteilung genutzt werden.

```
> IQ      <- c(112, 103, 87, 86, 90, 101, 90, 89, 122, 103)
> IQcls <- numeric(length(IQ))                                # neuen Vektor erstellen
> IQcls[IQ <= 100]           <- 1                          # Intervall bis inkl. 100
> IQcls[(IQ > 100) & (IQ <= 115)] <- 2                  # Intervall (100, 115]
> IQcls[IQ > 115]          <- 3                          # Intervall größer 115
> IQcls
[1] 2 2 1 1 1 2 1 1 3 2

# Klasseneinteilung der IQ-Werte mit recode()
> library(car)                                              # für recode()
> recode(IQ, "0:100=1; 101:115=2; else=3")
[1] 2 2 1 1 1 2 1 1 3 2
```

```
# Dichotomisierung mit ifelse()
> ifelse(IQ >= 100, "hi", "lo")
[1] "hi" "hi" "lo" "lo" "lo" "hi" "lo" "lo" "hi" "hi"
```

Besonders leicht lassen sich quantitative Variablen zudem mit `cut()` diskretisieren (vgl. Abschn. 2.6.7). Hierdurch werden sie zu Faktoren, dem Gegenstand des folgenden Abschnitts.

2.6 Gruppierungsfaktoren

Die Klasse `factor` existiert, um die Eigenschaften kategorialer Variablen abzubilden. Sie eignet sich insbesondere für Gruppierungsfaktoren im versuchsplanerischen Sinn. Ein Objekt dieser Klasse nimmt die jeweilige Gruppenzugehörigkeit von Beobachtungsobjekten auf und enthält Informationen darüber, welche Stufen die Variable prinzipiell umfasst. Für den Gebrauch in inferenzstatistischen Analysefunktionen ist es wichtig, dass Gruppierungsvariablen auch tatsächlich die Klasse `factor` besitzen. Insbesondere bei numerisch codierter Gruppenzugehörigkeit besteht sonst die Gefahr der Verwechslung mit echten quantitativen Variablen, was etwa bei linearen Modellen (z. B. Regression oder Varianzanalyse) für falsche Ergebnisse sorgen kann.

2.6.1 Ungeordnete Faktoren

Als Beispiel für eine Gruppenzugehörigkeit soll das qualitative Merkmal Geschlecht dienen, dessen Ausprägungen in einer Stichprobe zunächst als `character` Werte eingegeben und in einem Vektor gespeichert werden.

```
> sex <- c("m", "f", "f", "m", "m", "m", "f", "f")
```

Um den aus Zeichen bestehenden Vektor zu einem Gruppierungsfaktor mit zwei Ausprägungen zu machen, dient der Befehl `factor()`.

```
> factor(x=<Vektor>, levels=<Stufen>, labels=levels)
```

Unter `x` ist der umzuwandelnde Vektor einzutragen. Welche Stufen der Faktor prinzipiell annehmen kann, bestimmt das Argument `levels`. In der Voreinstellung werden die Faktorstufen automatisch anhand der in `x` tatsächlich vorkommenden Werte mit `sort(unique(x))` bestimmt.

```
> (sexFac <- factor(sex))
[1] m f f m m m f f
Levels: f m
```

Da die in `x` gespeicherten empirischen Ausprägungen nicht notwendigerweise auch alle theoretisch möglichen Kategorien umfassen müssen, kann an `levels` auch ein Vektor mit allen möglichen Stufen übergeben werden.

```
# 2 und 5 kommen nicht vor, sollen aber mögliche Ausprägungen sein
> factor(c(1, 1, 3, 3, 4, 4), levels=1:5)
[1] 1 1 3 3 4 4
Levels: 1 2 3 4 5
```

Die Stufenbezeichnungen stimmen in der Voreinstellung mit den `levels` überein, sie können aber auch durch einen für das Argument `labels` übergebenen Vektor umbenannt werden. Dies könnte etwa sinnvoll sein, wenn die Faktorstufen in einem Vektor numerisch codiert sind, im Faktor aber inhaltlich aussagekräftigere Namen erhalten sollen.

```
> (sexNum <- rbinom(10, size=1, prob=0.5))      # 0=Mann, 1=Frau
[1] 0 1 1 0 1 0 0 0 1 1

> factor(sexNum, labels=c("man", "woman"))
[1] man woman woman man woman man man man woman woman
Levels: man woman
```

Wenn die Stufenbezeichnungen eines Faktors im nachhinein geändert werden sollen, so kann dem von `levels(<Faktor>)` ausgegebenen Vektor ein Vektor mit passend vielen neuen Namen zugewiesen werden.

```
> levels(sexFac) <- c("female", "male")          # vorherige Stufen: f, m
> sexFac
[1] male female female male male male female female
Levels: female male
```

Die Anzahl der Stufen eines Faktors wird mit `nlevels(<Faktor>)` ausgegeben; wie häufig jede Stufe vorkommt, erfährt man durch `summary(<Faktor>)`.

```
> nlevels(sexFac)
[1] 2

> summary(sexFac)
female male
4      4
```

Die im Faktor gespeicherten Werte werden intern auf zwei Arten repräsentiert – zum einen mit den Namen der Faktorstufen in einem `character` Vektor im Attribut `levels`, zum anderen mit einer internen Codierung der Stufen über fortlaufende natürliche Zahlen, die der (ggf. alphabetischen) Reihenfolge der Ausprägungen entspricht. Dies wird in der Ausgabe der internen Struktur eines Faktors mit `str(<Faktor>)` deutlich. Die Namen der Faktorstufen werden mit `levels(<Faktor>)` ausgegeben, die interne numerische Repräsentation mit `unclass(<Faktor>)`.¹⁶

¹⁶Trotz dieser Codierung können Faktoren keinen mathematischen Transformationen unterzogen werden. Wenn die Namen der Faktorstufen aus Zahlen gebildet werden, kann es zu Diskrepanzen zwischen den Stufen und der internen Codierung kommen: `unclass(factor(10:15))` ergibt 1 2 3 4 5 6. Dies ist bei der üblichen Verwendung von Faktoren aber irrelevant.

```
> levels(sexFac)
[1] "female" "male"

> str(sexFac)
Factor w/ 2 levels "female","male": 2 1 1 2 2 2 1 1

> unclass(sexFac)
[1] 2 1 1 2 2 2 1 1

attr(,"levels")
[1] "female" "male"
```

2.6.2 Faktoren kombinieren

Faktoren lassen sich nicht wie Vektoren mit `c()` aneinanderhängen, da `c()` die Attribute der übergebenen Argumente (bis auf die Elementnamen) entfernt. Im Fall von Faktoren wären deren Klasse `factor` und die Stufen `levels` davon betroffen. Stattdessen muss ein komplizierterer Weg gewählt werden: Zunächst sind aus den zu kombinierenden Faktoren alle Ausprägungen in Form von `character` Vektoren zu extrahieren und diese dann mit `c()` zu verbinden, bevor aus dem kombinierten Vektor wieder ein Faktor erstellt wird. Dabei gehen allerdings mögliche Faktorstufen verloren, die nicht auch als Ausprägung tatsächlich vorkommen.

```
> (fac1 <- factor(sample(LETTERS, 5)))      # Faktor 1
[1] Q A P U J
Levels: A J P Q U

> (fac2 <- factor(sample(letters, 3)))       # Faktor 2
[1] z g t
Levels: g t z

> (charVec1 <- levels(fac1)[fac1])          # character Vektor zu Faktor 1
[1] "Q" "A" "P" "U" "J"

> (charVec2 <- levels(fac2)[fac2])          # character Vektor zu Faktor 2
[1] "z" "g" "t"

# Faktor der aneinandergehängten character Vektoren
> factor(c(charVec1, charVec2))
[1] Q A P U J z g t
Levels: A g J P Q t U z
```

Gilt es, lediglich einen bereits bestehenden Faktor zu vervielfachen, eignet sich dagegen wie bei Vektoren `rep()`.

```
> rep(fac1, times=2)                         # Faktor 1 wiederholen
[1] Q A P U J Q A P U J
Levels: A J P Q U
```

In Situationen, in denen sich experimentelle Bedingungen aus der Kombination von zwei oder mehr Faktoren ergeben, ist es bisweilen nützlich, die mehrfaktorielle in eine geeignete einfaktorielle Struktur zu überführen. Dabei werden alle Kombinationen von Faktorstufen als Stufen eines neuen einzelnen Faktors betrachtet – etwa im Kontext einer assoziierten einfaktoriellen Varianzanalyse bei einem eigentlich zweifaktoriellen Design (vgl. Abschn. 7.5). Dies ist mit `interaction()` möglich.

```
> interaction(<Faktor1>, <Faktor2>, ..., drop=FALSE)
```

Als Argument sind zunächst mehrere Faktoren gleicher Länge zu übergeben, die die Gruppenzugehörigkeit derselben Beobachtungsobjekte bzgl. verschiedener Gruppierungsfaktoren codieren. Auf ihren Stufenkombinationen basieren die Ausprägungen des neuen Faktors. Dabei kann es vorkommen, dass nicht für jede Stufenkombination Beobachtungen vorhanden sind, da einige Zellen im Versuchsdesign leer sind, oder kein vollständig gekreuztes Design vorliegt. In der Voreinstellung `drop=FALSE` erhält der neue Faktor auch für leere Zellen eine passende Faktorstufe. Mit `drop=TRUE` werden zu leeren Zellen gehörende Stufen dagegen weggelassen.

```
> (IV1 <- factor(rep(c("lo", "hi"), each=6)))      # Faktor 1
[1] lo lo lo lo lo hi hi hi hi hi
Levels: hi lo

> (IV2 <- factor(rep(1:3, times=4)))              # Faktor 2
[1] 1 2 3 1 2 3 1 2 3 1 2 3
Levels: 1 2 3

> interaction(IV1, IV2)                            # assoziiertes einfaktorielles Design
[1] lo.1 lo.2 lo.3 lo.1 lo.2 lo.3 hi.1 hi.2 hi.3 hi.1 hi.2 hi.3
Levels: hi.1 lo.1 hi.2 lo.2 hi.3 lo.3
```

2.6.3 Faktorstufen nachträglich ändern

Einem bestehenden Faktor können nicht beliebige Werte als Element hinzugefügt werden, sondern lediglich solche, die einer bereits existierenden Faktorstufe entsprechen. Bei Versuchen, einem Faktorelement einen anderen Wert zuzuweisen, wird das Element auf `NA` gesetzt und eine Warnung ausgegeben. Die Menge möglicher Faktorstufen kann jedoch über `levels()` erweitert werden, ohne dass es bereits zugehörige Beobachtungen gäbe.

```
> (status <- factor(c("hi", "lo", "hi")))
[1] hi lo hi
Levels: hi lo

# bisher nicht existierende Faktorstufe "mid"
> status[4] <- "mid"
> status
Warning message:
In `<-factor`(`*tmp*`, 4, value = "mid") :
invalid factor level, NAs generated
```

```
[1] hi lo hi <NA>
Levels: hi lo

> levels(status) <- c(levels(status), "mid")      # Stufe "mid" hinzufügen
> status[4] <- "mid"                                # neue Beobachtung "mid"
> status
[1] hi lo hi mid
Levels: hi lo mid
```

Stufen eines bestehenden Faktors lassen sich nicht ohne weiteres löschen. Die erste Möglichkeit, um einen gegebenen Faktor in einen Faktor mit weniger möglichen Stufen umzuwandeln, besteht im Zusammenfassen mehrerer ursprünglicher Stufen zu einer gemeinsamen neuen Stufe. Hierzu muss dem von `levels()` ausgegebenen Objekt eine Liste zugewiesen werden, die nach dem Muster `list(<neueStufe>=c("<alteStufe1>", "<alteStufe2>", ...))` aufgebaut ist (vgl. Abschn. 3.1). Alternativ eignet sich die in Abschn. 2.5.6 vorgestellte `recode()` Funktion aus dem Paket `car`, mit der sich auch Faktoren umcodieren lassen.

```
# kombiniere Stufen "mid" und "lo" zu "notHi", "hi" bleibt unverändert
> hiNotHi <- status                                     # Kopie des Faktors
> levels(hiNotHi) <- list(hi="hi", notHi=c("mid", "lo"))
> hiNotHi
[1] hi notHi hi notHi
Levels: hi notHi

> library(car)                                         # für recode()
> (statNew <- recode(status, "hi='high'; c('mid', 'lo')='notHigh'"))
[1] high notHigh high notHigh
Levels: high notHigh
```

Sollen dagegen Beobachtungen samt ihrer Stufen gelöscht werden, muss eine Teilmenge der Elemente des Faktors ausgegeben werden, die nicht alle Faktorstufen enthält. Zunächst umfasst diese Teilmenge jedoch noch wie vor alle ursprünglichen Stufen, wie in der Ausgabe unter `Levels` deutlich wird. Sollen nur die in der gewählten Teilmenge tatsächlich auftretenden Ausprägungen auch mögliche `Levels` sein, kann dies mit `droplevels()` erreicht werden.

```
> status[1:2]
[1] hi lo
Levels: hi lo mid

> (newStatus <- droplevels(status[1:2]))
[1] hi lo
Levels: hi lo
```

2.6.4 Geordnete Faktoren

Besteht eine Reihenfolge in den Stufen eines Gruppierungsfaktors i. S. einer ordinalen Variable, so lässt sich dieser Umstand mit der Funktion `ordered(<Vektor>, levels)` abbilden, die einen geordneten Gruppierungsfaktor erstellt. Dabei muss die inhaltliche Reihenfolge im Argument `levels` explizit angegeben werden, weil R sonst die Reihenfolge selbst bestimmt und ggf. die alphabetische heranzieht. Für die Elemente geordneter Faktoren sind die üblichen Ordnungsrelationen $<$, \leq , $>$, \geq definiert.

```
> (ordStat <- ordered(status, levels=c("lo", "mid", "hi")))
[1] hi lo hi mid
Levels: lo < mid < hi

> ordStat[1] > ordStat[2]                                # hi > lo?
[1] TRUE
```

Manche Funktionen zur inferenzstatistischen Analyse nehmen bei geordneten Faktoren Gleichabständigkeit in dem Sinne an, dass die inhaltliche Unterschiedlichkeit zwischen zwei benachbarten Stufen immer dieselbe ist. Trifft dies nicht zu, sollte im Zweifel auf ungeordnete Faktoren zurückgegriffen werden.

2.6.5 Reihenfolge von Faktorstufen bestimmen

Beim Sortieren von Faktoren wie auch in manchen statistischen Analysefunktionen ist die Reihenfolge der Faktorstufen bedeutsam. Werden die Faktorstufen beim Erstellen eines Faktors explizit mit `levels` oder `labels` angegeben, bestimmt die Reihenfolge dieser Elemente die Reihenfolge der Stufen. Ohne Verwendung von `levels` oder `labels` ergibt sich die Reihenfolge aus den sortierten Elementen des Vektors, der zum Erstellen des Faktors verwendet wird.¹⁷

```
> vec <- c(3, 4, 3, 2, 1, 4, 1, 1)

# ohne levels/labels -> Reihenfolge aus sortierten Elementen des Vektors
> factor(vec)
[1] 3 4 3 2 1 4 1 1
Levels: 1 2 3 4

# nur levels angegeben -> levels bestimmt Reihenfolge der Stufen
> factor(vec, levels=c(4, 3, 2, 1))
[1] 3 4 3 2 1 4 1 1
Levels: 4 3 2 1

# levels und labels angegeben -> labels bestimmt Reihenfolge der Stufen
> factor(vec, levels=1:4, labels=c("one", "two", "three", "four"))
[1] three four three two one four one one
Levels: one two three four
```

¹⁷Sind dessen Elemente Zeichenketten mit numerischer Bedeutung, so ist zu beachten, dass die Reihenfolge dennoch alphabetisch bestimmt wird – die Stufe "10" käme demnach vor der Stufe "4".

Um die Reihenfolge der Stufen nachträglich zu ändern, kann ein Faktor in einen geordneten Faktor unter Verwendung des `levels` Arguments ungewandelt werden (s.o.). Als weitere Möglichkeit wird mit `relevel(<Faktor>, ref="Referenzstufe")` die für `ref` übergebene Faktorstufe zur ersten Stufe des Faktors. `reorder()` ändert die Reihenfolge der Faktorstufen ebenfalls nachträglich. Die Stufen werden so geordnet, dass ihre Reihenfolge durch die gruppenweise gebildeten empirischen Kennwerte einer Variable bestimmt ist.¹⁸

```
> reorder(x=<Faktor>, X=<Vektor>, FUN=<Funktion>)
```

Als erstes Argument `x` wird der Faktor mit den zu ordnenden Stufen erwartet. Für das Argument `X` ist ein numerischer Vektor derselben Länge wie `x` zu übergeben, der auf Basis von `x` in Gruppen eingeteilt wird. Pro Gruppe wird die mit `FUN` bezeichnete Funktion angewendet, die einen Vektor zu einem skalaren Kennwert verarbeiten muss. Als Ergebnis werden die Stufen von `x` entsprechend der Kennwerte geordnet, die sich aus der gruppenweisen Anwendung von `FUN` ergeben (vgl. Abschn. 2.7.10 für `tapply()`).

```
> fac1 <- factor(rep(LETTERS[1:3], each=10))
> vec  <- rnorm(30, rep(c(10, 5, 15), each=10), 3)
> reorder(fac1, vec, FUN=mean)
[1] A A A A A A A A B B B B B B B B C C
[23] C C C C C C C C
Levels: B < A < C
```

Kontrolle: Mittelwerte pro Gruppe

```
> tapply(vec, fac1, FUN=mean)
      A          B          C 
10.18135  6.47932 13.50108
```

Beim Sortieren von Faktoren wird die Reihenfolge der Elemente durch die Reihenfolge der Faktorstufen bestimmt, die nicht mit der numerischen oder alphabetischen Reihenfolge der Stufenbezeichnungen übereinstimmen muss. Damit kann das Sortieren eines Faktors zu einem anderen Ergebnis führen als das Sortieren eines Vektors, auch wenn diese oberflächlich dieselben Elemente enthalten.

```
> (fac2 <- factor(sample(1:2, 10, replace=TRUE), labels=c("B", "A")))
[1] A A A B B A B B B B
Levels: B A
```

```
> sort(fac2)
[1] B B B B B A A A A
```

```
> sort(as.character(fac2))
[1] "A" "A" "A" "A" "B" "B" "B" "B" "B" "B"
```

¹⁸Das Paket `DescTools` stellt zudem die Funktion `reorder.factor()` bereit, mit der Faktorstufen beliebig geordnet werden können.

2.6.6 Faktoren nach Muster erstellen

Da Faktoren im Zuge der Einteilung von Beobachtungsobjekten in Gruppen oft nach festem Muster erstellt werden müssen, lassen sich Faktoren als Alternative zur manuellen Anwendung von `rep()` und `factor()` mit `gl()` automatisiert erzeugen.

```
> gl(n=<Stufen>, k=<Zellbesetzung>, labels=1:n, ordered=FALSE)
```

Das Argument `n` gibt die Anzahl der Stufen an, die der Faktor besitzen soll. Mit `k` wird festgelegt, wie häufig jede Faktorstufe realisiert werden soll, wie viele Beobachtungen also jede Bedingung umfasst. Für `labels` kann ein Vektor mit so vielen Gruppenbezeichnungen angegeben werden, wie Stufen vorhanden sind. In der Voreinstellung werden die Gruppen numeriert. Um einen geordneten Faktor zu erstellen, ist `ordered=TRUE` zu setzen.

```
> (fac1 <- factor(rep(c("A", "B"), c(5, 5)))) # manuell
[1] A A A A A B B B B B
Levels: A B
```

```
> (fac2 <- gl(2, 5, labels=c("less", "more"), ordered=TRUE))
[1] less less less less more more more more more
Levels: less < more
```

Sollen die Elemente des Faktors in eine zufällige Reihenfolge gebracht werden, um die Zuordnung von Beobachtungsobjekten zu Gruppen zu randomisieren, kann dies wie in Abschn. 2.5.2 beschrieben geschehen.

```
> sample(fac2, length(fac2), replace=FALSE)
[1] more more less more less less less more more less
Levels: less < more
```

Bei mehreren Faktoren mit vollständig gekreuzten Faktorstufen kann `expand.grid()` verwendet werden, um alle Stufenkombinationen zu erstellen (vgl. Abschn. 2.3.3). Dabei ist die angestrebte Gruppenbesetzung pro Zelle nur bei einem der hier im Aufruf durch `gl()` erstellten Faktoren anzugeben, beim anderen ist sie auf 1 zu setzen. Das Ergebnis ist ein Datensatz (vgl. Abschn. 3.2).

```
> expand.grid(IV1=gl(2, 2, labels=c("a", "b")), IV2=gl(3, 1))
  IV1  IV2
1    a    1
2    a    1
3    b    1
4    b    1
5    a    2
6    a    2
7    b    2
8    b    2
9    a    3
10   a    3
11   b    3
12   b    3
```

2.6.7 Quantitative in kategoriale Variablen umwandeln

Aus den in einem Vektor gespeicherten Werten einer quantitativen Variable lässt sich mit `cut()` ein Gruppierungsfaktor erstellen – die quantitative Variable wird so in eine kategoriale umgewandelt (vgl. auch Abschn. 2.5.7). Dazu muss zunächst der Wertebereich des Vektors in disjunkte Intervalle eingeteilt werden. Die einzelnen Werte werden dann entsprechend ihrer Zugehörigkeit zu diesen Intervallen Kategorien zugeordnet und erhalten als Wert die zugehörige Faktorstufe.

```
> cut(x=<Vektor>, breaks=<Intervalle>, labels=<Bezeichnungen>, ordered=FALSE)
```

Die Intervalle werden über das Argument `breaks` festgelegt. Hier ist entweder die Anzahl der (dann gleich breiten) Intervalle anzugeben, oder die Intervallgrenzen selbst als Vektor. Die Intervalle werden in der Form $(a, b]$ gebildet, sind also nach unten offen und nach oben geschlossen. Anders gesagt ist die untere Grenze nicht Teil des Intervalls, die obere schon. Die Unter- und Obergrenze des insgesamt möglichen Wertebereichs müssen bei der Angabe von `breaks` berücksichtigt werden, ggf. sind dies `-Inf` und `Inf` für negativ und positiv unendliche Werte.

Wenn die Faktorstufen andere Namen als die zugehörigen Intervallgrenzen tragen sollen, können sie über das Argument `labels` explizit angegeben werden. Dabei ist darauf zu achten, dass die Reihenfolge der neuen Benennungen der Reihenfolge der gebildeten Intervalle entspricht. Soll es sich im Ergebnis um einen geordneten Faktor handeln, ist `ordered=TRUE` zu setzen.

```
> IQ      <- rnorm(100, mean=100, sd=15)
> IQfac <- cut(IQ, breaks=c(0, 85, 115, Inf), labels=c("lo", "mid", "hi"))
> IQfac[1:5]
[1] hi lo mid mid mid lo
Levels: lo mid hi
```

Um annähernd gleich große Gruppen zu erhalten, können für die Intervallgrenzen bestimmte Quantile der Daten gewählt werden, etwa der Median für den Median-Split (vgl. Abschn. 2.7.3).

```
> medSplit <- cut(IQ, breaks=c(-Inf, median(IQ), Inf))
> summary(medSplit)           # Kontrolle: Häufigkeiten der Kategorien
medSplit
(-Inf,97.6]  (97.6,Inf]
      50          50
```

Für mehr als zwei etwa gleich große Gruppen lässt sich die Ausgabe von `quantile()` (vgl. Abschn. 2.7.5) direkt an das Argument `breaks` übergeben. Dies ist möglich, da `quantile()` neben den Quantilen auch das Minimum und Maximum der Werte ausgibt. Damit das unterste Intervall auch das Minimum einschließt – und nicht wie alle übrigen Intervalle nach unten offen ist, muss das Argument `include.lowest=TRUE` gesetzt werden.

```
# 4 ähnliche große Gruppen, unterstes Intervall dabei unten geschlossen
> IQdiscr <- cut(IQ, quantile(IQ), include.lowest=TRUE)
> summary(IQdiscr)           # Kontrolle: Häufigkeiten der Kategorien
IQdiscr
```

[62.1,87.1]	(87.1,97.6]	(97.6,107]	(107,154]
25	25	25	25

Anstelle von `cut()` kann auch `CutQ()` aus dem Paket `DescTools` verwendet werden, um eine Variable in Intervalle mit etwa gleichen Häufigkeiten zu unterteilen.

2.7 Deskriptive Kennwerte numerischer Daten

Die deskriptive Beschreibung von Variablen ist ein wichtiger Teil der Analyse empirischer Daten, die gemeinsam mit der grafischen Darstellung (vgl. Abschn. 14.6) hilft, ihre Struktur besser zu verstehen. Die hier umgesetzten statistischen Konzepte und Techniken werden als bekannt vorausgesetzt und finden sich in vielen Lehrbüchern der Statistik (Eid, Gollwitzer & Schmitt, 2013).

R stellt für die Berechnung aller gängigen Kennwerte separate Funktionen bereit, die meist erwarten, dass die Daten in Vektoren gespeichert sind.¹⁹ Es sei an dieser Stelle daran erinnert, dass sich logische Wahrheitswerte ebenfalls in einem numerischen Kontext verwenden lassen, wobei der Wert `TRUE` wie eine 1, der Wert `FALSE` wie eine 0 behandelt wird.

Mit `summary(<Vektor>)` können die wichtigsten deskriptiven Kennwerte einer Datenreihe abgerufen werden – dies sind Minimum, erstes Quartil, Median, Mittelwert, drittes Quartil und Maximum. Die Ausgabe ist ein Vektor mit benannten Elementen.

```
> age <- c(17, 30, 30, 25, 23, 21)
> summary(age)
  Min.  1st Qu.  Median   Mean  3rd Qu.  Max.
17.00    21.50   24.00  24.33   28.75   30.00
```

2.7.1 Summen, Differenzen und Produkte

Mit `sum(<Vektor>)` wird die Summe aller Elemente eines Vektors berechnet. Die kumulierte Summe erhält man mit `cumsum(<Vektor>)`.

```
> sum(age)
[1] 146

> cumsum(age)
[1] 17 47 77 102 125 146
```

Um die Differenzen aufeinander folgender Elemente eines Vektors (also eines Wertes zu einem Vorgänger) zu berechnen, kann die Funktion `diff(<Vektor>, lag=1)` verwendet werden. Mit ihrem Argument `lag` wird kontrolliert, über welchen Abstand die Differenz gebildet wird. Die Voreinstellung `1` bewirkt, dass die Differenz eines Wertes zum unmittelbar vorhergehenden berechnet wird. Die Ausgabe umfasst `lag` Werte weniger, als der Vektor Elemente besitzt.

¹⁹Viele von ihnen werden in `Desc()` aus dem Paket `DescTools` integriert.

```
> diff(age)
[1] 13 0 -5 -2 -2

> diff(age, lag=2)
[1] 13 -5 -7 -4
```

Das Produkt aller Elemente eines Vektors wird mit `prod(<Vektor>)` berechnet, das kumulierte Produkt mit `cumprod(<Vektor>)`.²⁰ `factorial(<Zahl>)` ermittelt die Fakultät $n!$ einer Zahl n .²¹

```
> prod(age)
[1] 184747500

> cumprod(age)
[1] 17 510 15300 382500 8797500 184747500

> factorial(5)
[1] 120
```

2.7.2 Extremwerte

Mit `min(<Vektor>)` und `max(<Vektor>)` können die Extremwerte eines Vektors erfragt werden. `range(<Vektor>)` gibt den größten und kleinsten Wert zusammengefasst als Vektor aus.

```
> max(age)                      # Maximum
[1] 30

> range(c(17, 30, 30, 25, 23, 21))
[1] 17 30
```

Den Index des größten bzw. kleinsten Wertes liefern `which.max(<Vektor>)` bzw. `which.min(<Vektor>)`. Die letztgenannte Funktion lässt sich etwa nutzen, um herauszufinden, welches Element eines Vektors am nächsten an einem vorgegebenen Wert liegt.

```
> which.min(age)                 # Position des Minimums
[1] 1

> vec <- c(-5, -8, -2, 10, 9)   # Vektor
> val <- 0                        # Referenzwert
> which.min(abs(vec-val))        # welches Element liegt am nächsten an 0?
[1] 3
```

²⁰Bei Gleitkommazahlen kumulieren sich bei wiederholter Multiplikation Rundungsfehler, die durch die interne Darstellungsart solcher Zahlen unvermeidlich sind (vgl. Abschn. 1.3.6). Numerisch stabiler als `prod(<Vektor>)` ist deswegen u.U. die Rücktransformation der Summe der logarithmierten Werte mit `exp(sum(log(<Vektor>)))` als Umsetzung von $\exp\left(\sum_i \ln x_i\right)$ – vorausgesetzt x ist echt positiv. `prod(numeric(0))` ist gleich 1.

²¹Für natürliche Zahlen gilt $n! = \Gamma(n + 1)$, in R als `gamma(<Zahl> + 1)` berechenbar.

Um die Spannweite (*range*) von Werten eines Vektors, also die Differenz von kleinstem und größtem Wert zu ermitteln, ist `diff()` nützlich.

```
> diff(range(c(17, 30, 30, 25, 23, 21)))
[1] 13
```

Die Funktionen `pmin(Vektor1, Vektor2, ...)` und `pmax(Vektor1, Vektor2, ...)` vergleichen zwei oder mehr Vektoren elementweise hinsichtlich der Größe der in ihnen gespeicherten Werte. Sie liefern einen Vektor aus den pro Position größten bzw. kleinsten Werten zurück.

```
> vec1 <- c(5, 2, 0, 7)
> vec2 <- c(3, 3, 9, 2)
> pmax(vec1, vec2)
[1] 5 3 9 7

> pmin(vec1, vec2)
[1] 3 2 0 2
```

2.7.3 Mittelwert, Median und Modalwert

Mit `mean(x=Vektor)` wird das arithmetische Mittel $\frac{1}{n} \sum_i x_i$ eines Vektors x der Länge n berechnet.²²

```
> age <- c(17, 30, 30, 25, 23, 21)
> mean(age)
[1] 24.33333
```

Für die Berechnung eines gewichteten Mittels, bei dem die Gewichte nicht wie bei `mean()` für alle Werte identisch sind ($\frac{1}{n}$), eignet sich die Funktion `weighted.mean(x=Vektor, w=Gewichte)`. Ihr zweites Argument `w` muss ein Vektor derselben Länge wie `x` sein und die Gewichte benennen. Der Einfluss jedes Wertes auf den Mittelwert ist gleich dem Verhältnis seines Gewichts zur Summe aller Gewichte.

```
> weights <- c(0.6, 0.6, 0.3, 0.2, 0.4, 0.6)
> weighted.mean(age, weights)
[1] 23.70370
```

Um beim geometrischen Mittel $\frac{1}{n} \prod_i x_i$ sich fortsetzende Rundungsfehler zu vermeiden, eignet sich für positive x_i `exp(mean(log(x)))` als Umsetzung von $e^{\frac{1}{n} \sum_i \ln x_i}$ (vgl. Fußnote 20). Das harmonische Mittel $n / \sum_i \frac{1}{x_i}$ berechnet sich als Kehrwert des Mittelwertes der Kehrwerte (für $x_i \neq 0$), also mit `1 / mean(1/x)`. Alternativ stellt das Paket `DescTools` die Funktionen `Gmean()` und `Hmean()` bereit.

`median(x=Vektor)` gibt den Median, also das 50%-Quantil einer empirischen Verteilung aus. Dies ist der Wert, für den die empirische kumulative Häufigkeitsverteilung von `x` erstmalig

²²Hier ist zu beachten, dass `x` tatsächlich ein etwa mit `c(...)` gebildeter Vektor ist: Der Aufruf `mean(1, 7, 3)` gibt anders als `mean(c(1, 7, 3))` nicht den Mittelwert der Daten 1, 7, 3 aus. Stattdessen ist die Ausgabe gleich dem ersten übergebenen Argument.

mindestens den Wert 0.5 erreicht (vgl. Abschn. 2.10.6), der also $\geq 50\%$ (und $\leq 50\%$) der Werte ist. Im Fall einer geraden Anzahl von Elementen in `x` wird zwischen den beiden mittleren Werten von `sort(<Vektor>)` gemittelt, andernfalls das mittlere Element von `sort(<Vektor>)` ausgegeben.

```
> sort(age)
[1] 17 21 23 25 30 30

> median(age)
[1] 24

> ageNew <- c(age, 22)
> sort(ageNew)
[1] 17 21 22 23 25 30 30

> median(ageNew)
[1] 23
```

Für die Berechnung des Modalwertes, also des am häufigsten vorkommenden Wertes eines Vektors, stellt der Basisumfang von R keine separate Funktion bereit. Es kann aber auf die Funktion `mlv(<Vektor>, method="mfv")` aus dem Paket `modeest` (Poncet, 2012) ausgewichen werden. Bei mehreren Werten gleicher Häufigkeit interpoliert sie zwischen diesen.

```
> vec <- c(11, 22, 22, 33, 33, 33, 33)      # häufigster Wert: 33
> library(modeest)                          # für mlv()
> mlv(vec, method="mfv")
Mode (most likely value): 33
Bickel's modal skewness: -0.4285714
```

Eine manuelle Alternative bietet `table()` zur Erstellung von Häufigkeitstabellen (vgl. Abschn. 2.10). Die folgende Methode gibt zunächst den Index des Maximums der Häufigkeitstabelle aus. Den Modalwert erhält man zusammen mit seiner Auftretenshäufigkeit durch Indizieren der mit `unique()` gebildeten Einzelwerte des Vektors mit diesem Index.

```
> (tab <- table(vec))                      # Häufigkeitstabelle
vec
11 22 33
 1   2   4

> (modIdx <- which.max(tab))                # Index des Modalwerts
33
 3

> unique(vec)[modIdx]                      # Modalwert selbst
[1] 33
```

2.7.4 Robuste Maße der zentralen Tendenz

Wenn Daten Ausreißer aufweisen, kann dies den Mittelwert stark verzerrn, so dass er den Lageparameter der zugrundeliegenden Variable nicht mehr gut repräsentiert. Aus diesem Grund existieren Maße der zentralen Tendenz von Daten, die weniger stark durch einzelne extreme Werte beeinflusst werden.

Der gestützte Mittelwert wird ohne einen bestimmten Anteil an Extremwerten berechnet. Mit dem Argument `mean(x, trim=<Zahl>)` wird der gewünschte Anteil an Extremwerten aus der Berechnung des Mittelwerts ausgeschlossen. `<Zahl>` gibt dabei den Anteil der Werte an, der auf jeder der beiden Seiten der empirischen Verteilung verworfen werden soll. Um etwa den Mittelwert ohne die extremen 5% der Daten zu berechnen, ist `trim=0.025` zu setzen.

```
> age <- c(17, 30, 30, 25, 23, 21)
> mean(age)                                # Vergleich: Mittelwert
[1] 24.33333

> mean(age, trim=0.2)                      # gestützter Mittelwert
[1] 24.75
```

Bei der Winsorisierung von Daten mit `Winsorize()` aus dem Paket `DescTools` wird ein bestimmter Anteil an Extremwerten auf beiden Seiten der Verteilung durch jeweils den letzten Wert ersetzt, der noch nicht als Extremwert gilt. Dafür legt das Argument `probs` in Form eines Vektors die beiden Quantile als Grenzen fest. Der übliche Mittelwert dieser Daten ist dann der winsorisierte Mittelwert.

```
> library(DescTools)                         # für Winsorize()
> (ageWins <- Winsorize(age, probs=c(0.2, 0.8))) # winsorisierte Daten
[1] 21 30 30 25 23 21

> mean(ageWins)                            # winsor. Mittelwert
[1] 25
```

Der Hodges-Lehmann-Schätzer des Lageparameters einer Variable berechnet sich als Median der $n(n + 1)/2$ vielen Mittelwerte aller Wertepaare (x_i, x_j) der Daten mit sich selbst (mit $j \geq i$). Vergleiche hierfür `HodgesLehmann()` aus dem Paket `DescTools` sowie `wilcox.test()` in Abschn. 10.5.3.

```
> library(DescTools)                         # für HodgesLehmann()
> HodgesLehmann(age, conf.level=0.95)
est.(pseudo)median   lwr.ci    upr.ci
                23.99998  20.99997  27.50005

# manuelle Kontrolle -> Mittelwert aller Wertepaare
> pairM <- outer(age, age, FUN="+") / 2
> median(pairM[lower.tri(pairM, diag=TRUE)])      # deren Median
[1] 24
```

Für Huber M -Schätzer der Lage von Verteilungen existiert `huberM()` aus dem Paket `robustbase` (Rousseeuw et al., 2014). Als Schätzer für die Differenz der Lageparameter von zwei Variablen wird der Hodges-Lehmann-Schätzer als Median aller $n_1 \cdot n_2$ paarweisen Differenzen der Werte aus beiden Datenreihen gebildet (vgl. Abschn. 10.5.4).

2.7.5 Prozentrang, Quartile und Quantile

Angewendet auf Vektoren mit logischen Werten lassen sich mit `sum()` Elemente zählen, die eine bestimmte Bedingung erfüllen. So kann der Prozentrang eines Wertes als prozentualer Anteil derjenigen Werte ermittelt werden, die nicht größer als er sind (vgl. auch Abschn. 2.10.6).

```
> age <- c(17, 30, 30, 25, 23, 21)
> sum(age <= 25)                                # wie viele Elemente <= 25?
[1] 4

> 100 * (sum(age <= 25) / length(age))        # Prozentrang von 25
[1] 66.66667
```

Mit `quantile(x=<Vektor>, probs=seq(0, to=1, by=0.25))` werden in der Voreinstellung die Quartile eines Vektors bestimmt. Dies sind jene Werte, die größer oder gleich einem ganzzahligen Vielfachen von 25% der Datenwerte und kleiner oder gleich den Werten des verbleibenden Anteils der Daten sind. Das erste Quartil ist etwa $\geq 25\%$ (und $\leq 75\%$) der Daten. Das Ergebnis von `quantile()` ist ein Vektor mit benannten Elementen.

```
> (quant <- quantile(age))
  0%    25%    50%    75%   100%
17.00  21.50  24.00  28.75  30.00

> quant[c("25%", "50%")]
  25%  50%
21.5  24.0
```

Über das `probs=<Vektor>` Argument können statt der Quartile auch andere Anteile eingegeben werden, deren Wertegrenzen gewünscht sind. Zur Berechnung der Werte, die einen bestimmten Anteil der Daten abschneiden, wird ggf. zwischen den in `x` tatsächlich vorkommenden Werten interpoliert.²³

```
> vec <- sample(seq(0, to=1, by=0.01), 1000, replace=TRUE)
> quantile(vec, probs=c(0, 0.2, 0.4, 0.6, 0.8, 1))
  0%    20%    40%    60%    80%   100%
0.000  0.190  0.400  0.604  0.832  1.000
```

²³Zur Berechnung von Quantilen stehen verschiedene Rechenwege zur Verfügung, vgl. `?quantile`.

2.7.6 Varianz, Streuung, Schiefe und Wölbung

Mit `var(x=⟨Vektor⟩)` wird die korrigierte Varianz $s^2 = \frac{1}{n-1} \sum_i (x_i - \bar{x})^2$ zur erwartungstreuen Schätzung der Populationsvarianz auf Basis einer Variable X der Länge n mit Mittelwert \bar{x} ermittelt.²⁴ Die Umrechnungsformel zur Berechnung der unkorrigierten Varianz $S^2 = \frac{1}{n} \sum_i (x_i - \bar{x})^2$ aus der korrigierten lautet $S^2 = \frac{n-1}{n} s^2$.²⁵

```
> age <- c(17, 30, 30, 25, 23, 21)          # Daten
> N   <- length(age)                      # Anzahl Beobachtungen
> M   <- mean(age)                        # Mittelwert

> var(age)                                # korrigierte Varianz
[1] 26.26667

> sum((age-M)^2) / (N-1)                  # manuelle Berechnung
[1] 26.26667

> ((N-1) / N) * var(age)                 # unkorrigierte Varianz
[1] 21.88889

> sum((age-M)^2) / N                     # manuelle Berechnung
[1] 21.88889
```

Die korrigierte Streuung s kann durch Ziehen der Wurzel aus der korrigierten Varianz oder mit `sd(x=⟨Vektor⟩)` berechnet werden. Auch hier basiert das Ergebnis auf der bei der Varianz erläuterten Korrektur zur Schätzung der Populationsstreuung auf Basis einer empirischen Stichprobe. Die Umrechnungsformel zur Berechnung der unkorrigierten Streuung $S = \sqrt{\frac{1}{n} \sum_i (x_i - \bar{x})^2}$ aus der korrigierten lautet $S = \sqrt{\frac{n-1}{n}} s$.

```
> sqrt(var(age))                         # Wurzel aus Varianz
[1] 5.125102

> sd(age)                                # korrigierte Streuung
[1] 5.125102

> sqrt((N-1) / N) * sd(age)            # unkorrigierte Streuung
[1] 4.678556

> sqrt(sum((age-M)^2) / N)              # manuelle Berechnung
[1] 4.678556
```

Schiefe und Wölbung (Kurtosis) als höhere zentrale Momente empirischer Verteilungen lassen sich mit `Skew()` und `Kurt()` aus dem Paket `DescTools` ermitteln.

²⁴Für ein Diversitätsmaß kategorialer Daten vgl. Abschn. 2.7.7 und für robuste Varianzschätzer Abschn. 2.7.9.

²⁵Als Alternative lässt sich `cov.wt()` verwenden (vgl. Abschn. 2.8.10).

2.7.7 Diversität kategorialer Daten

Der Diversitätsindex H ist ein Streuungsmaß für kategoriale Daten, der mit Werten im Intervall $[0, 1]$ angibt, wie sehr sich die Daten auf die Kategorien verteilen. Sind f_j die relativen Häufigkeiten der p Kategorien (vgl. Abschn. 2.10), ist $H = -\frac{1}{\ln p} \sum (f_j \cdot \ln f_j)$. Für $f_j = 0$ ist $f_j \cdot \ln f_j$ dabei als 0 definiert. H ist genau dann 0, wenn die Daten konstant sind, also nur eine von mehreren Kategorien auftritt. Für eine Gleichverteilung, in der alle Kategorien dieselbe Häufigkeit besitzen, die Daten also maximal verteilt sind, ist $H = 1$. R verfügt über keine spezialisierte Funktion für die Berechnung von H . Es lässt sich aber der Umstand nutzen, dass H bis auf den Faktor $\frac{1}{\ln p}$ mit dem Shannon-Index aus der Informationstheorie übereinstimmt, der sich mit der aus dem Paket `DescTools` stammenden Funktion `Entropy()` berechnen lässt. Dafür ist das Argument `base` auf die Eulersche Zahl e zu setzen.²⁶ Die Funktion erwartet als Argument den Vektor der absoluten oder relativen Häufigkeiten der Kategorien.

```
# Faktor inkl. einer nicht besetzten Stufe "Q"
> fac <- factor(c("C", "D", "A", "D", "E", "D", "C", "E", "E", "B", "E"),
+                 levels=c(LETTERS[1:5], "Q"))

> P   <- nlevels(fac)                                # Anzahl Kategorien
> (Fj <- prop.table(table(fac)))                     # relative Häufigkeiten
      A          B          C          D          E          Q
0.09090909 0.09090909 0.18181818 0.27272727 0.36363636 0.00000000

> library(DescTools)                               # für Entropy()
> shannonIdx <- Entropy(Fj, base=exp(1))        # Shannon Index
> (H <- (1/log(P)) * shannonIdx)                # Diversität
[1] 0.8193845

> keep <- Fj > 0                                 # Indizes Häufigk. > 0
> -(1/log(P)) * sum(Fj[keep] * log(Fj[keep]))    # Kontrolle ...
```

2.7.8 Kovarianz und Korrelation

Mit `cov(x=<Vektor1>, y=<Vektor2>)` wird die korrigierte Kovarianz $\frac{1}{n-1} \sum_i ((x_i - \bar{x}) \cdot (y_i - \bar{y}))$ zweier Variablen X und Y derselben Länge n berechnet. Die unkorrigierte Kovarianz muss nach der bereits für die Varianz genannten Umrechnungsformel ermittelt werden (vgl. Abschn. 2.7.6, Fußnote 25).

```
> x <- c(17, 30, 30, 25, 23, 21)                  # Daten Variable 1
> y <- c(1, 12, 8, 10, 5, 3)                      # Daten Variable 2
> cov(x, y)                                       # korrigierte Kovarianz
[1] 19.2

> N   <- length(x)                                # Anzahl Objekte
```

²⁶Zudem gilt folgende Beziehung zur diskreten Kullback-Leibler-Divergenz KL_{eq} der beobachteten Häufigkeiten zur Gleichverteilung: $H = -\frac{1}{\ln p} KL_{eq} + 1$.

```
> Mx <- mean(x)                      # Mittelwert Var 1
> My <- mean(y)                      # Mittelwert Var 2
> sum((x-Mx) * (y-My)) / (N-1)    # korrigierte Kovarianz manuell
[1] 19.2

> ((N-1) / N) * cov(x, y)          # unkorrig. Kovarianz aus korrigierter
[1] 16

> sum((x-Mx) * (y-My)) / N        # unkorrigierte Kovarianz manuell
[1] 16
```

Neben der voreingestellten Berechnungsmethode für die Kovarianz nach Pearson kann auch die Rang-Kovarianz nach Spearman oder Kendall berechnet werden (vgl. Abschn. 10.3.1).

Analog zur Kovarianz kann mit `cor(x=(Vektor1), y=(Vektor2))` die herkömmliche Produkt-Moment-Korrelation $r_{XY} = \frac{\text{Kov}_{XY}}{s_X \cdot s_Y}$ oder die Rangkorrelation berechnet werden.²⁷ Für die Korrelation gibt es keinen Unterschied beim Gebrauch von korrigierten und unkorrigierten Streuungen, so dass sich nur ein Kennwert ergibt.

```
> cor(x, y)                         # Korrelation
[1] 0.8854667

> cov(x, y) / (sd(x) * sd(y))     # manuelle Berechnung
[1] 0.8854667
```

Für die Berechnung der Partialkorrelation zweier Variablen X und Y ohne eine dritte Variable Z kann die Formel $r_{(XY).Z} = \frac{r_{XY} - (r_{XZ} \cdot r_{YZ})}{\sqrt{(1-r_{XZ}^2) \cdot (1-r_{YZ}^2)}}$ umgesetzt werden, da die Basisinstallation von R hierfür keine eigene Funktion bereitstellt.²⁸ Für eine alternative Berechnungsmethode, die sich die Eigenschaft der Partialkorrelation als Korrelation der Residuen der Regressionen von X auf Z und Y auf Z zunutze macht, vgl. Abschn. 6.7.

```
> NN <- 100
> zz <- runif(NN)
> xx <- zz + rnorm(NN, 0, 0.5)
> yy <- zz + rnorm(NN, 0, 0.5)
> (cor(xx, yy) - (cor(xx, zz)*cor(yy, zz))) /
+   sqrt((1-cor(xx, zz)^2) * (1-cor(yy, zz)^2))
[1] 0.0753442
```

Die Semipartialkorrelation einer Variable Y mit einer Variable X ohne eine dritte Variable Z unterscheidet sich von der Partialkorrelation dadurch, dass nur von X der i.S. der linearen Regression durch Z aufklärbare Varianzanteil auspartialisiert wird, nicht aber von Y . Die

²⁷Das Paket `polycor` (Fox, 2010) beinhaltet Funktionen für die polychorische und polyseriale Korrelation zur Schätzung der latenten Korrelation von künstlich in Kategorien eingeteilten Variablen, die eigentlich stetig sind. Für die multiple Korrelation i.S. der Wurzel aus dem Determinationskoeffizienten R^2 in der multiplen linearen Regression vgl. Abschn. 6.2.2. Die kanonische Korrelation zweier Gruppen von Variablen, die an denselben Beobachtungsobjekten erhoben wurden, ermittelt `cancor()`.

²⁸Wohl aber das Paket `DescTools` mit `PartCor()`.

Semipartialkorrelation berechnet sich durch $r_{(X.Z)Y} = \frac{r_{XY} - (r_{XZ} \cdot r_{YZ})}{\sqrt{1 - r_{XZ}^2}}$, oder als Korrelation von Y mit den Residuen der Regression von X auf Z (vgl. Abschn. 6.7).

```
> (cor(xx, yy) - (cor(xx, zz) * cor(yy, zz))) / sqrt(1-cor(xx, zz)^2)
[1] 0.06275869
```

2.7.9 Robuste Streuungsmaße und Kovarianzschätzer

Ausreißer können gewöhnliche Streuungen und Kovarianzen in dem Sinne stark verzerrn, dass sie nicht mehr gut die Eigenschaften der zugrundeliegenden Verteilung der beteiligten Variablen repräsentieren. Aus diesem Grund existieren Streuungsmaße und Kovarianzschätzer, die weniger stark durch einzelne extreme Werte beeinflusst werden.

Mit `IQR(<Vektor>)` wird der Interquartilabstand erfragt, also die Differenz von drittem und erstem Quartil.

```
> age <- c(17, 30, 30, 25, 23, 21)
> sd(age)                                # Vergleich: gewöhnliche Streuung
[1] 5.125102

> quantile(age)                          # Quartile
  0%   25%   50%   75%   100%
17.00 21.50 24.00 28.75 30.00

> IQR(age)                               # Interquartilabstand
[1] 7.25
```

Die mittlere absolute Abweichung vom Mittelwert oder Median ist manuell oder über `MeanAD()` aus dem Paket `DescTools` zu berechnen. Dagegen steht für den Median der absoluten Abweichungen vom Median `mad(<Vektor>)` bereit.²⁹

```
> mean(abs(age - mean(age)))    # mittlere abs. Abweichung vom Mittelwert
[1] 4

> mean(abs(age-median(age)))    # mittlere absolute Abweichung vom Median
[1] 4

> mad(age)                      # Median der abs. Abweichungen vom Median
[1] 6.6717
```

Bei der Winsorisierung von Daten mit `Winsorize()` aus dem Paket `DescTools` wird ein bestimmter Anteil an Extremwerten auf beiden Seiten der Verteilung durch jeweils den letzten Wert ersetzt, der noch nicht als Extremwert gilt. Dafür legt das Argument `probs` in Form eines

²⁹ Die Funktion multipliziert den eigentlichen Median der absoluten Abweichungen mit dem Faktor 1.4826, der über das Argument `constant` auf einen anderen Wert gesetzt werden kann. Der Faktor ist so gewählt, dass der Kennwert bei normalverteilten Variablen asymptotisch mit der Streuung übereinstimmt, da für standardnormalverteilte Variablen X gilt: $E(MAD(X)) = 0.6745 = \frac{1}{1.4826}$.

Vektors die beiden Quantile als Grenzen fest. Die übliche Varianz dieser Daten ist dann die winsorisierte Varianz.

```
> library(DescTools)                                # für Winsorize()
> ageWins <- Winsorize(age, probs=c(0.2, 0.8))    # winsorisierte Daten
> var(ageWins)                                     # winsorisierte Varianz
[1] 17.2
```

Weitere robuste Streuungsschätzer stellt das Paket **robustbase** mit **Sn()**, **Qn()** und **scaleTau2()** bereit. Für robuste Schätzer einer theoretischen Kovarianzmatrix vgl. **covOGK()** und **covMcd()** aus demselben Paket. Die mittlere absolute Differenz nach Gini kann mit **GiniMd()** aus dem Paket **rms** (Harrell Jr, 2014b) ermittelt werden.

2.7.10 Kennwerte getrennt nach Gruppen berechnen

Oft sind die Werte einer in verschiedenen Bedingungen erhobenen Variable in einem Vektor **x** gespeichert, wobei sich die zu jedem Wert gehörende Beobachtungsbedingung aus einem Faktor oder der Kombination mehrerer Faktoren ergibt. Jeder Faktor besitzt dabei dieselbe Länge wie **x** und codiert die Zugehörigkeit der Beobachtungen in **x** zu den Stufen einer Gruppierungsvariable. Dabei müssen nicht für jede Bedingung auch gleich viele Beobachtungen vorliegen.

Sollen Kennwerte von **x** jeweils getrennt für jede Bedingung bzw. Kombination von Bedingungen berechnet werden, können **ave()** und **tapply()** herangezogen werden.

```
> ave(x=<Vektor>, <Faktor1>, <Faktor2>, ...,      FUN=<Funktion>)
> tapply(X=<Vektor>, INDEX=<Liste mit Faktoren>, FUN=<Funktion>, ...)
```

Als Argumente werden neben dem zuerst zu nennenden Datenvektor die Faktoren übergeben. Bei **ave()** geschieht dies einfach in Form mehrerer durch Komma getrennter Gruppierungsfaktoren. Bei **tapply()** müssen die Faktoren in einer Liste zusammengefasst werden, deren Komponenten die einzelnen Faktoren sind (vgl. Abschn. 3.1). Mit dem Argument **FUN** wird schließlich die pro Gruppe auf die Daten anzuwendende Funktion angegeben. Der Argumentname **FUN** ist bei **ave()** immer zu nennen, andernfalls wäre nicht ersichtlich, dass kein weiterer Faktor gemeint ist. In der Voreinstellung von **ave()** wird **mean()** angewendet.

In der Ausgabe von **ave()** wird jeder Einzelwert von **x** durch den für die entsprechende Gruppe berechneten Kennwert ersetzt, was etwa in der Berechnung von Quadratsummen linearer Modelle Anwendung finden kann (vgl. Abschn. 7.3.6).

Im Beispiel sei ein IQ-Test mit Personen durchgeführt worden, die aus einer Treatment- (**T**), Wartelisten- (**WL**) oder Kontrollgruppe (**CG**) stammen. Weiterhin sei das Geschlecht als Faktor berücksichtigt worden.

```
> Njk    <- 2                                # Zellbesetzung
> P      <- 2                                # Anzahl Stufen Geschlecht
> Q      <- 3                                # Anzahl Stufen Treatment
> sex    <- factor(rep(c("f", "m"), times=Q*Njk))
> group  <- factor(rep(c("T", "WL", "CG"), each=P*Njk))
> table(sex, group)                          # Zellbesetzungen
```

```

group
sex CG T WL
f   2  2  2
m   2  2  2

> IQ <- round(rnorm(Njk*P*Q, mean=100, sd=15))
> ave(IQ, sex, FUN=mean)           # Mittelwerte nach Geschlecht
[1] 96.83333 100.33333 96.83333 100.33333 96.83333 100.33333 96.83333
[8] 100.33333 96.83333 100.33333 96.83333 100.33333

```

Die Ausgabe von `tapply()` dient der Übersicht über die gruppenweise berechneten Kennwerte, wobei das Ergebnis ein Objekt der Klasse `array` ist (vgl. Abschn. 2.9). Bei einem einzelnen Gruppierungsfaktor ist dies einem benannten Vektor ähnlich und bei zweien einer zweidimensionalen Kreuztabelle (vgl. Abschn. 2.10).

```

> (tapRes <- tapply(IQ, group, FUN=mean))      # Mittelwert pro Gruppe
    CG      T      WL
94.25 99.75 101.75

# Mittelwert pro Bedingungskombination
> tapply(IQ, list(sex, group), FUN=mean)
    CG      T      WL
f  98.0  91.0 101.5
m  90.5 108.5 102.0

```

Wenn das Ergebnis ein eindimensionales array ist, lässt sich auch `tapply()` dazu verwenden, um jeden Wert durch einen für seine Gruppe berechneten Kennwert zu ersetzen: Die Elemente der Ausgabe tragen dann als Namen die zugehörigen Gruppenbezeichnungen und lassen sich über diese Namen indizieren. Die als Indizes verwendbaren Gruppenbezeichnungen finden sich im Faktor, wobei jeder Index entsprechend der zugehörigen Gruppengröße mehrfach auftaucht.

```

> tapRes[group]
    T      T      T      T      WL      WL      WL      WL      CG      CG      CG
99.75 99.75 99.75 99.75 101.75 101.75 101.75 101.75 94.25 94.25 94.25 94.25

```

Da für `FUN` beliebige Funktionen an `tapply()` übergeben werden können, muss man sich nicht darauf beschränken, für Gruppen getrennt einzelne Kennwerte zu berechnen. Genauso sind Funktionen zugelassen, die pro Gruppe mehr als einen einzelnen Wert ausgeben, wobei das Ergebnis von `tapply()` dann eine Liste ist (vgl. Abschn. 3.1): Jede Komponente der Liste beinhaltet die Ausgabe von `FUN` für eine Gruppe.

So ist es etwa auch möglich, sich die Werte jeder Gruppe selbst ausgeben zu lassen, indem man die Funktion `identity()` verwendet, die ihr Argument unverändert ausgibt (vgl. Abschn. 3.3.5 für `split()`).

```

> tapply(IQ, sex, FUN=identity)      # IQ-Werte pro Geschlecht
$f
[1] 100 82 88 115 86 110

```

```
$m
[1] 120 97 97 107 80 101

> split(IQ, sex)                      # Kontrolle ...
> IQ[sex == "f"]                       # Kontrolle ...
> IQ[sex == "m"]                       # Kontrolle ...
```

2.7.11 Funktionen auf geordnete Paare von Werten anwenden

Eine Verallgemeinerung der Anwendung einer Funktion auf jeden Wert eines Vektors stellt die Anwendung einer Funktion auf alle geordneten Paare aus den Werten zweier Vektoren dar.

```
> outer(X=<Vektor1>, Y=<Vektor2>, FUN="*", ...)
```

Für die Argumente X und Y ist dabei jeweils ein Vektor einzutragen, unter FUN eine Funktion, die zwei Argumente in Form von Vektoren verarbeiten kann.³⁰ Da Operatoren nur Funktionen mit besonderer Schreibweise sind, können sie hier ebenfalls eingesetzt werden, müssen dabei aber in Anführungszeichen stehen (vgl. Abschn. 1.2.5, Fußnote 14). Voreinstellung ist die Multiplikation, für diesen Fall existiert auch die Kurzform in Operatorschreibweise X %o% Y. Sollen an FUN weitere Argumente übergeben werden, kann dies an Stelle der ... geschehen, wobei mehrere Argumente durch Komma zu trennen sind. Die Ausgabe erfolgt in Form einer Matrix (vgl. Abschn. 2.8).

Als Beispiel sollen alle Produkte der Zahlen 1–5 als Teil des kleinen 1×1 ausgegeben werden.

```
> outer(1:5, 1:5, FUN="*")
     [,1]   [,2]   [,3]   [,4]   [,5]
[1,]    1      2      3      4      5
[2,]    2      4      6      8     10
[3,]    3      6      9     12     15
[4,]    4      8     12     16     20
[5,]    5     10     15     20     25
```

2.8 Matrizen

Wenn für jedes Beobachtungsobjekt Daten von mehreren Variablen vorliegen, könnten die Werte jeder Variable in einem separaten Vektor gespeichert werden. Eine andere Möglichkeit zur gemeinsamen Repräsentation aller Variablen bieten Objekte der Klasse **matrix** als Spezialfall von arrays (vgl. Abschn. 2.9).³¹

³⁰Vor dem Aufruf von FUN verlängert **outer()** X und Y so, dass beide die Länge **length(X)*length(Y)** besitzen und sich aus der Kombination der Elemente mit gleichem Index alle geordneten Paare ergeben.

³¹Eine Matrix ist in R zunächst nur eine rechteckige Anordnung von Werten und nicht mit dem gleichnamigen mathematischen Konzept zu verwechseln. Wie **attributes(<Matrix>)** zeigt, sind Matrizen intern lediglich Vektoren mit einem Attribut (vgl. Abschn. 1.3), das Auskunft über die Dimensionierung der Matrix, also die Anzahl ihrer Zeilen und Spalten liefert. Eine Matrix kann deshalb maximal so viele Werte speichern, wie ein Vektor Elemente besitzen kann (vgl. Abschn. 2.1.1, Fußnote 1). Für Rechenoperationen mit Matrizen im Kontext der linearen Algebra vgl. Abschn. 12.1.

```
> matrix(data=<Vektor>, nrow=<Anzahl>, ncol=<Anzahl>, byrow=FALSE)
```

Unter `data` ist der Vektor einzutragen, der alle Werte der zu bildenden Matrix enthält. Mit `nrow` wird die Anzahl der Zeilen dieser Matrix festgelegt, mit `ncol` die der Spalten. Die Länge des Vektors muss gleich dem Produkt von `nrow` und `ncol` sein, das gleich der Zahl der Zellen ist. Mit dem auf `FALSE` voreingestellten Argument `byrow` wird die Art des Einlesens der Daten aus dem Vektor in die Matrix bestimmt – es werden zunächst die Spalten nacheinander gefüllt. Mit `byrow=TRUE` werden die Werte über die Zeilen eingelesen.

```
> age <- c(17, 30, 30, 25, 23, 21)
> matrix(age, nrow=3, ncol=2, byrow=FALSE)
 [,1]  [,2]
[1,]    17    25
[2,]    30    23
[3,]    30    21

> (ageMat <- matrix(age, nrow=2, ncol=3, byrow=TRUE))
 [,1]  [,2]  [,3]
[1,]    17    30    30
[2,]    25    23    21
```

2.8.1 Datentypen in Matrizen

Wie Vektoren können Matrizen verschiedene Datentypen besitzen, etwa `numeric`, wenn sie Zahlen beinhalten, oder `character` im Fall von Zeichenketten. Jede einzelne Matrix kann dabei aber ebenso wie ein Vektor nur einen einzigen Datentyp haben, alle Matrixelemente müssen also vom selben Datentyp sein.³² Fügt man einer numerischen Matrix eine Zeichenkette als Element hinzu, so werden die numerischen Matrixelemente automatisch in Zeichenketten umgewandelt, was an den hinzugekommenen Anführungszeichen zu erkennen ist.³³ Auf die ehemals numerischen Werte können dann keine Rechenoperationen mehr angewendet werden. Dieser Umstand macht Matrizen letztlich weniger geeignet für empirische Datensätze, für die stattdessen Objekte der Klasse `data.frame` bevorzugt werden sollten (vgl. Abschn. 3.2).³⁴

2.8.2 Dimensionierung, Zeilen und Spalten

Die Dimensionierung einer Matrix (die Anzahl ihrer Zeilen und Spalten) liefert die Funktion `dim(<Matrix>)`, die auch auf arrays (vgl. Abschn. 2.9) oder Datensätze (vgl. Abschn. 3.2) anwendbar ist. Sie gibt einen Vektor aus, der die Anzahl der Zeilen und Spalten in dieser Reihenfolge als Elemente besitzt. Über `nrow(<Matrix>)` und `ncol(<Matrix>)` kann die Anzahl der Zeilen bzw. Spalten auch einzeln ausgegeben werden.

³²Indem einer Liste (vgl. Abschn. 3.1) eine Dimensionierung als Attribut hinzugefügt wird, lässt sich diese Einschränkung indirekt umgehen: `mat <- list(1, "X")`; `dim(mat) <- c(2, 1)` erstellt eine heterogene Matrix.

³³Allgemein gesprochen werden alle Elemente in den umfassendsten Datentyp umgewandelt, der notwendig ist, um alle Werte ohne Informationsverlust zu speichern (vgl. Abschn. 1.3.5).

³⁴Da Matrizen numerisch effizienter als Objekte der Klasse `data.frame` verarbeitet werden können, sind sie dagegen bei der Analyse sehr großer Datenmengen vorzuziehen.

```
> age      <- c(17, 30, 30, 25, 23, 21)
> ageMat <- matrix(age, nrow=2, ncol=3, byrow=FALSE)
> dim(ageMat)                      # Dimensionierung
[1] 2 3

> nrow(ageMat)                    # Anzahl der Zeilen
[1] 2

> ncol(ageMat)                   # Anzahl der Spalten
[1] 3

> prod(dim(ageMat))             # Anzahl der Elemente
[1] 6
```

Eine Matrix wird mit `t(<Matrix>)` transponiert, wodurch ihre Zeilen zu den Spalten der Transponierten und entsprechend ihre Spalten zu Zeilen der Transponierten werden.

```
> t(ageMat)
[,1] [,2]
[1,] 17   30
[2,] 30   25
[3,] 23   21
```

Wird ein Vektor über `as.matrix(<Vektor>)` in eine Matrix umgewandelt, entsteht als Ergebnis eine Matrix mit einer Spalte und so vielen Zeilen, wie der Vektor Elemente enthält.

```
> as.matrix(1:3)
[,1]
[1,] 1
[2,] 2
[3,] 3
```

Um eine Matrix in einen Vektor umzuwandeln, sollte entweder `as.vector(<Matrix>)` oder einfach `c(<Matrix>)` verwendet werden.³⁵ Die Anordnung der Elemente entspricht dabei dem Aneinanderhängen der Spalten der Matrix.

```
> c(ageMat)
[1] 17 30 30 25 23 21
```

Mitunter ist es nützlich, zu einer gegebenen Matrix zwei zugehörige Matrizen zu erstellen, in denen jedes ursprüngliche Element durch seinen Zeilen- bzw. Spaltenindex ersetzt wurde. Dieses Ziel lässt sich mit den `row(<Matrix>)` und `col(<Matrix>)` erreichen.

```
> P <- 2                      # Anzahl der Zeilen
> Q <- 3                      # Anzahl der Spalten
> (pqMat <- matrix(1:(P*Q), nrow=P, ncol=Q))
[,1] [,2] [,3]
```

³⁵`c()` entfernt die Attribute der übergebenen Argumente bis auf ihre Elementnamen. Matrizen verlieren damit ihre Dimensionierung `dim` und ihre Klasse `matrix`.

```
[1,]    1    3    5
[2,]    2    4    6

> (rowMat <- row(pqMat))          # Zeilenindizes
[1,] [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2

> (colMat <- col(pqMat))          # Spaltenindizes
[1,] [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
```

Die so gewonnenen Matrizen können etwa dazu verwendet werden, eine dreispartige Matrix zu erstellen, die in einer Spalte alle Elemente der ursprünglichen Matrix besitzt und in den anderen beiden Spalten die zugehörigen Zeilen- und Spaltenindizes enthält (vgl. Abschn. 2.8.5).

```
> cbind(rowIdx=c(rowMat), colIdx=c(colMat), val=c(pqMat))
      rowIdx colIdx val
[1,]    1    1    1
[2,]    2    1    2
[3,]    1    2    3
[4,]    2    2    4
[5,]    1    3    5
[6,]    2    3    6
```

Eine andere Anwendungsmöglichkeit besteht darin, logische untere bzw. obere Dreiecksmatrizen zu erstellen, die auch von `lower.tri()` und `upper.tri()` erzeugt werden können.

```
> mat <- matrix(sample(1:10, 16, replace=TRUE), nrow=4, ncol=4)
> col(mat) >= row(mat)          # obere Dreiecksmatrix
      [,1] [,2] [,3] [,4]
[1,] TRUE  TRUE  TRUE  TRUE
[2,] FALSE TRUE  TRUE  TRUE
[3,] FALSE FALSE TRUE  TRUE
[4,] FALSE FALSE FALSE TRUE
```

2.8.3 Elemente auswählen und verändern

In einer Matrix ist es ähnlich wie bei einem Vektor möglich, sich einzelne Elemente mit dem `[<Zeile>, <Spalte>]` Operator anzeigen zu lassen. Der erste Index in der eckigen Klammer gibt dabei die Zeile des gewünschten Elements an, der zweite dessen Spalte.³⁶

```
> ageMat[2, 2]
[1] 25
```

³⁶Für Hilfe zu diesem Thema vgl. `?Extract`.

Analog zum Vorgehen bei Vektoren können auch bei Matrizen einzelne Elemente unter Angabe ihres Zeilen- und Spaltenindex durch die Zuweisung eines Wertes verändert werden. Fehlt bei Zuweisungen der Index `(Matrix)[]`, werden alle Elemente der Matrix ersetzt. Wenn der zugewiesene Vektor dabei weniger Elemente als die Matrix besitzt, wird er automatisch passend verlängert (vgl. Abschn. 2.5.4).

```
> ageMat[2, 2] <- 24
> ageMat[2, 2]
[1] 24

> ageMatCopy <- ageMat           # zu verändernde Kopie der Matrix
> ageMatCopy[] <- c(1, 2, 3)      # Veränderung mit zykl. Verlängerung
> ageMatCopy
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    2    1    3
```

Man kann sich Zeilen oder Spalten auch vollständig ausgeben lassen. Dafür wird für die vollständig aufzulistende Dimension kein Index eingetragen, jedoch das Komma trotzdem gesetzt. Es können dabei auch beide Dimensionen weggelassen werden, was für die Ausgabe denselben Effekt wie das gänzliche Weglassen des `[<Index>]` Operators hat.

```
> ageMat[2, ]          # Werte 2. Zeile
[1] 30 24 21

> ageMat[ , 1]         # Werte 1. Spalte
[1] 17 30

> ageMat[ , ]           # gesamte Matrix, äquivalent: ageMat[]
     [,1] [,2] [,3]
[1,]    17    30    23
[2,]    30    24    21
```

Bei der Ausgabe einer einzelnen Zeile oder Spalte wird diese automatisch in einen Vektor umgewandelt, verliert also eine Dimension. Möchte man dies – wie es häufig der Fall ist – verhindern, kann beim `[<Index>]` Operator als weiteres Argument `drop=FALSE` angegeben werden. Das Ergebnis ist dann eine Matrix mit nur einer Zeile oder Spalte.³⁷

```
> ageMat[ , 1, drop=FALSE]
     [,1]
[1,]    17
[2,]    30
```

Analog zum Vorgehen bei Vektoren können auch gleichzeitig mehrere Matrixelemente ausgewählt und verändert werden, indem man etwa eine Sequenz oder einen anderen Vektor als Indexvektor für eine Dimension festlegt.

³⁷Dagegen bleibt `ageMat[FALSE, FALSE]` eine leere Matrix mit 0 Zeilen und 0 Spalten.

```

> ageMat[, 2:3]                                # 2. und 3. Spalte
  [,1]  [,2]
[1,] 30   23
[2,] 24   21

> ageMat[, c(1, 3)]                            # 1. und 3. Spalte
  [,1]  [,2]
[1,] 17   23
[2,] 30   21

> ageMatNew <- ageMat
> (replaceMat <- matrix(c(11, 21, 12, 22), nrow=2, ncol=2))
  [,1]  [,2]
[1,] 11   12
[2,] 21   22

> ageMatNew[, c(1, 3)] <- replaceMat          # ersetze Spalten 1 und 3
> ageMatNew
  [,1]  [,2]  [,3]
[1,] 11   30   12
[2,] 21   24   22

```

2.8.4 Weitere Wege, Elemente auszuwählen und zu verändern

Auf Matrixelemente kann auch zugegriffen werden, wenn nur ein einzelner Index genannt wird. Die Matrix wird dabei implizit in den Vektor der untereinander gehängten Spalten umgewandelt.

```

> idxVec <- c(1, 3, 4)
> ageMat[idxVec]
[1] 17 30 24

```

Weiter können Matrizen auch durch eine logische Matrix derselben Dimensionierung – etwa das Ergebnis eines logischen Vergleichs – indiziert werden, die für jedes Element bestimmt, ob es ausgegeben werden soll. Das Ergebnis ist ein Vektor der ausgewählten Elemente.

```

> (idxMatLog <- ageMat >= 25)
  [,1]  [,2]  [,3]
[1,] FALSE  TRUE  FALSE
[2,] TRUE  FALSE  FALSE

> ageMat[idxMatLog]
[1] 30 30

```

Schließlich ist es möglich, eine zweispaltige numerische Indexmatrix zu verwenden, wobei jede Zeile dieser Matrix ein Element der indizierten Matrix auswählt – der erste Eintrag einer Zeile gibt den Zeilenindex, der zweite den Spaltenindex des auszuwählenden Elements an. Eine solche

Matrix entsteht etwa bei der Umwandlung einer logischen in eine numerische Indexmatrix mittels `which()` (vgl. Abschn. 2.2.2), wenn das Argument `arr.ind=TRUE` gesetzt ist. Auch hier ist das Ergebnis ein Vektor der ausgewählten Elemente.

```
> (idxMatNum <- which(idxMatLog, arr.ind=TRUE))
      row  col
[1,]    2    1
[2,]    1    2

> ageMat[idxMatNum]
[1] 30 30
```

`arrayInd(<Indexvektor>, dim(<Matrix>))` konvertiert einen numerischen Indexvektor, der eine Matrix im obigen Sinn als Vektor indiziert, in eine zweispaltige numerische Indexmatrix.

```
> (idxMat <- arrayInd(idxVec, dim(ageMat)))
      [,1] [,2]
[1,]    1    1
[2,]    1    2
[3,]    2    2
```

2.8.5 Matrizen verbinden

Die Funktionen `cbind(<Vektor1>, <Vektor2>, ...)` und `rbind(<Vektor1>, <Vektor2>, ...)` fügen Vektoren zu Matrizen zusammen. Das `c` bei `cbind()` steht für columns (Spalten), das `r` entsprechend für rows (Zeilen). In diesem Sinn werden die Vektoren mit `cbind()` spaltenweise nebeneinander, und mit `rbind()` zeilenweise untereinander angeordnet.

```
> age     <- c(19, 19, 19, 31, 24)
> weight <- c(95, 76, 76, 94, 76)
> height <- c(197, 179, 186, 189, 173)
> rbind(age, weight, height)
      [,1] [,2] [,3] [,4] [,5]
age     19    19   31   19   24
weight  95    76   94   76   76
height  197   178  189  184  173

> (mat <- cbind(age, weight, height))
      age  weight  height
[1,] 19     95     197
[2,] 19     76     178
[3,] 31     94     189
[4,] 19     76     184
[5,] 24     76     173
```

2.8.6 Matrizen sortieren

Die Zeilen von Matrizen können mit Hilfe von `order()` entsprechend der Reihenfolge der Werte in einer ihrer Spalten sortiert werden. Die Funktion `sort()` ist hier nicht anwendbar, ihr Einsatz ist auf Vektoren beschränkt.

```
> order(<Vektor>, partial, decreasing=FALSE)
```

Für `<Vektor>` ist die Spalte einer Datenmatrix einzutragen, deren Werte in eine Reihenfolge gebracht werden sollen. Unter `decreasing` wird die Sortierreihenfolge eingestellt: In der Voreinstellung `FALSE` wird aufsteigend sortiert, auf `TRUE` gesetzt absteigend. Die Ausgabe ist ein Indexvektor, der die Zeilenindizes der zu ordnenden Matrix in der Reihenfolge der Werte des Sortierkriteriums enthält (vgl. Abschn. 2.5.1).³⁸

```
> (rowOrder1 <- order(mat[, "age"])) # Kriterium: Alter
[1] 1 2 4 5 3
```

Soll die gesamte Matrix entsprechend der Reihenfolge dieser Variable angezeigt werden, ist der von `order()` ausgegebene Indexvektor zum Indizieren der Zeilen der Matrix zu benutzen. Dabei ist der Spaltenindex unter Beibehaltung des Kommas wegzulassen.

```
> mat[rowOrder1, ]
   age  weight  height
[1,] 19      95     197
[2,] 19      76     178
[3,] 19      76     184
[4,] 24      76     173
[5,] 31      94     189
```

Mit dem Argument `partial` kann noch eine weitere Matrixspalte eingetragen werden, die dann als sekundäres Sortierkriterium verwendet wird. So kann eine Matrix etwa zunächst hinsichtlich einer die Gruppenzugehörigkeit darstellenden Variable sortiert werden und dann innerhalb jeder Gruppe nach der Reihenfolge der Werte einer anderen Variable. Es können noch weitere Sortierkriterien durch Komma getrennt als Argumente vorhanden sein, es gibt also keine Beschränkung auf nur zwei solcher Kriterien.

```
# sortiere primär nach Alter und sekundär nach Gewicht
> rowOrder2 <- order(mat[, "age"], partial=mat[, "weight"])
> mat[rowOrder2, ]
   age  weight  height
[1,] 19      76     178
[2,] 19      76     184
[3,] 19      95     197
[4,] 24      76     173
[5,] 31      94     189
```

³⁸Die Funktion sortiert *stabil*: Zeilen mit gleich großen Werten des Sortierkriteriums behalten ihre Reihenfolge relativ zueinander bei, werden also beim Sortievorgang nicht zufällig vertauscht.

Das Argument `decreasing` legt global für alle Sortierkriterien fest, ob auf- oder absteigend sortiert wird. Soll die Sortierreihenfolge dagegen zwischen den Kriterien variieren, kann einzelnen numerischen Kriterien ein - vorangestellt werden, was als Umkehrung der mit `decreasing` eingestellten Reihenfolge zu verstehen ist.

```
# sortiere aufsteigend nach Gewicht und absteigend nach Größe
> rowOrder3 <- order(mat[, "weight"], -mat[, "height"])
> mat[rowOrder3, ]
  age  weight  height
[1,] 19      76     184
[2,] 19      76     178
[3,] 24      76     173
[4,] 31      94     189
[5,] 19      95     197
```

2.8.7 Randkennwerte berechnen

Die Summe aller Elemente einer numerischen Matrix lässt sich mit `sum(Matrix)`, die separat über jede Zeile oder jede Spalte gebildeten Summen durch `rowSums(Matrix)` bzw. `colSums(Matrix)` berechnen. Gleiches gilt für den Mittelwert aller Elemente, der mit der Funktion `mean(Matrix)` ermittelt wird, und für die mit `rowMeans()` bzw. `colMeans()` separat über jede Zeile oder jede Spalte berechneten Mittelwerte.

```
> sum(mat)                                # Summe aller Elemente
[1] 1450

> rowSums(mat)                            # Summen jeder Zeile
[1] 311 273 314 279 273

> mean(mat)                               # Mittelwert aller Elemente
[1] 96.66667

> colMeans(mat)                           # Mittelwerte jeder Spalte
  age  weight  height
22.4   83.4   184.2
```

2.8.8 Beliebige Funktionen auf Matrizen anwenden

Wenn eine andere Funktion als die Summe oder der Mittelwert separat auf jeweils jede Zeile oder jede Spalte angewendet werden soll, ist dies mit `apply()` zu erreichen.

```
> apply(X=<Matrix>, MARGIN=<Nummer>, FUN=<Funktion>, ...)
```

X erwartet die Matrix der zu verarbeitenden Daten. Unter MARGIN wird angegeben, ob die Funktion Kennwerte der Zeilen (1) oder Spalten (2) berechnet. Für FUN ist die anzuwendende Funktion einzusetzen, die als Argument einen Vektor akzeptieren muss. Gibt sie mehr als einen

Wert zurück, ist das Ergebnis eine Matrix mit den Rückgabewerten von FUN in den Spalten. Die drei Punkte ... stehen für optionale, ggf. durch Komma getrennte Argumente von FUN, die an diese Funktion weitergereicht werden.

```
> apply(mat, 2, sum)                      # Summen jeder Spalte
age   weight   height
112      417      921

> apply(mat, 1, max)                      # Maxima jeder Zeile
[1] 197 178 189 184 173

> apply(mat, 1, range)                    # Range jeder Zeile
[,1] [,2] [,3] [,4] [,5]
[1,] 19    19    31    19    24
[2,] 197   178   189   184   173

> apply(mat, 2, mean, trim=0.1)        # gestutzte Mittelwerte jeder Spalte
age   weight   height
22.4   83.4   184.2
```

Im letzten Beispiel wird das für ... eingesetzte Argument `trim=0.1` an `mean()` weitergereicht.

2.8.9 Matrix zeilen- oder spaltenweise mit Kennwerten verrechnen

Zeilen- und Spaltenkennwerte sind häufig Zwischenergebnisse, die für weitere Berechnungen mit einer Matrix nützlich sind. So ist es etwa zum spaltenweisen Zentrieren einer Matrix notwendig, von jedem Wert den zugehörigen Spaltenmittelwert abzuziehen. Anders gesagt soll die Matrix dergestalt mit einem Vektor verrechnet werden, dass auf jede Spalte dieselbe Operation (hier: Subtraktion), aber mit einem anderen Wert angewendet wird – nämlich mit dem Element des Vektors der Spaltenmittelwerte, das dieselbe Position im Vektor besitzt wie die Spalte in der Matrix. Die genannte Operation lässt sich mit `sweep()` durchführen.

```
> sweep(x=<Matrix>, MARGIN=<Nummer>, STATS=<Kennwerte>, FUN=<Funktion>, ...)
```

Das Argument `x` erwartet die Matrix der zu verarbeitenden Daten. Unter `MARGIN` wird angegeben, ob die Funktion jeweils Zeilen (1) oder Spalten (2) mit den Kennwerten verrechnet. An `STATS` sind diese Kennwerte in Form eines Vektors mit so vielen Einträgen zu übergeben, wie `x` Zeilen (`MARGIN=1`) bzw. Spalten (`MARGIN=2`) besitzt. Für `FUN` ist die anzuwendende Funktion einzusetzen, Voreinstellung ist die Subtraktion `"+"`. Die drei Punkte ... stehen für optionale, ggf. durch Komma getrennte Argumente von `FUN`, die an diese Funktion weitergereicht werden.

Im Beispiel sollen die Daten einer Matrix erst zeilenweise, dann spaltenweise zentriert werden.

```
> Mj <- rowMeans(mat)                      # Mittelwerte der Zeilen
> Mk <- colMeans(mat)                      # Mittelwerte der Spalten
> sweep(mat, 2, Mk, "-")                  # spaltenweise zentrieren
age   weight   height
[1,] -3.4     11.6    12.8
```

```
[2,] -3.4   -7.4   -6.2
[3,]  8.6   10.6    4.8
[4,] -3.4   -7.4   -0.2
[5,]  1.6   -7.4  -11.2

> scale(mat, center=TRUE, scale=FALSE)          # Kontrolle mit scale() ...
> sweep(mat, 1, Mj, "-")                      # zeilenweise zentrieren
      age     weight     height
[1,] -84.66667 -8.666667 93.33333
[2,] -72.00000 -15.000000 87.00000
[3,] -73.66667 -10.666667 84.33333
[4,] -74.00000 -17.000000 91.00000
[5,] -67.00000 -15.000000 82.00000

> t(scale(t(mat), center=TRUE, scale=FALSE))  # Kontrolle mit scale() ...
```

2.8.10 Kovarianz- und Korrelationsmatrizen

Zum Erstellen von Kovarianz- und Korrelationsmatrizen können die aus Abschn. 2.7.8 bekannten Funktionen `cov(<Matrix>)` und `cor(<Matrix>)` verwendet werden, wobei die Werte in Form einer Matrix mit Variablen in den Spalten übergeben werden müssen. `cov()` liefert die korrigierte Kovarianzmatrix.

```
> cov(mat)                                     # Kovarianzmatrix
      age     weight     height
age   27.80   22.55     0.4
weight 22.55  102.80    82.4
height  0.40   82.40   87.7

> cor(mat)                                    # Korrelationsmatrix
      age     weight     height
age   1.000000000  0.4218204  0.008100984
weight 0.421820411  1.0000000  0.867822404
height 0.008100984  0.8678224  1.000000000
```

`cov2cor(<K>)` wandelt eine korrigierte Kovarianzmatrix \mathbf{K} in eine Korrelationsmatrix um. Weiterhin existiert mit `cov.wt()` eine Funktion, die direkt die unkorrigierte Kovarianzmatrix ermitteln kann.

```
> cov.wt(x=<Matrix>, method=c("unbiased", "ML"))
```

Unter `x` ist die Matrix einzutragen, von deren Spalten paarweise die Kovarianz bestimmt werden soll. Um diese Funktion auf einen einzelnen Vektor anwenden zu können, muss dieser zunächst mit `as.matrix(<Vektor>)` in eine einspaltige Matrix konvertiert werden. Mit `method` lässt sich wählen, ob die korrigierten oder unkorrigierten Varianzen und Kovarianzen ausgerechnet werden – Voreinstellung ist "unbiased" für die korrigierten Kennwerte. Sind die unkorrigierten

Kennwerte gewünscht, ist "ML" zu wählen, da sie bei normalverteilten Variablen die Maximum-Likelihood-Schätzung der theoretischen Parameter auf Basis einer Stichprobe darstellen.

Das Ergebnis der Funktion ist eine Liste (vgl. Abschn. 3.1), die die Kovarianzmatrix als Komponente `cov`, die Mittelwerte als Komponente `center` und die Anzahl der eingegangenen Fälle als Komponente `n.obs` (number of observations) besitzt.

```
> cov.wt(mat, method="ML")
$cov
      age   weight   height
age    22.24   18.04    0.32
weight 18.04   82.24   65.92
height  0.32   65.92   70.16

$center
  age   weight   height
22.4    83.4   184.2

$n.obs
[1] 5
```

Mit `diag(<Matrix>)` lassen sich aus einer Kovarianzmatrix die in der Diagonale stehenden Varianzen extrahieren (vgl. Abschn. 12.1.1).

```
> diag(cov(mat))
  age   weight   height
27.8  102.8   87.7
```

Um gleichzeitig die Kovarianz oder Korrelation einer durch `<Vektor>` gegebenen Variable mit mehreren anderen, spaltenweise zu einer Matrix zusammengefassten Variablen zu berechnen, dient der Aufruf `cov(<Matrix>, <Vektor>)` bzw. `cor(<Matrix>, <Vektor>)`. Das Ergebnis ist eine Matrix mit so vielen Zeilen, wie das erste Argument Spalten besitzt.

```
> vec <- rnorm(nrow(mat))
> cor(mat, vec)
[,1]
age    -0.1843847
weight -0.6645798
height -0.6503452
```

2.9 Arrays

Das Konzept der Speicherung von Daten in eindimensionalen Vektoren und zweidimensionalen Matrizen lässt sich mit der Klasse `array` auf höhere Dimensionen verallgemeinern. In diesem Sinne sind Vektoren und Matrizen ein- bzw. zweidimensionale Spezialfälle von arrays, weshalb sich arrays in allen wesentlichen Funktionen auch wie Matrizen verhalten. So müssen die in

einem array gespeicherten Werte alle denselben Datentyp aufweisen, wie es auch bei Vektoren und Matrizen der Fall ist (vgl. Abschn. 2.8.1).³⁹

```
> array(data=<Vektor>, dim=length(data), dimnames=NULL)
```

Für `data` ist ein Datenvektor mit den Werten anzugeben, die das array speichern soll. Mit `dim` wird die Dimensionierung festgelegt, also die Anzahl der Dimensionen und der Werte pro Dimension. Dies geschieht mit Hilfe eines Vektors, der pro Dimension ein Element beinhaltet, das die Anzahl der zugehörigen Werte spezifiziert. Das Argument `dim=c(2, 3, 4)` würde etwa festlegen, dass das array zwei Zeilen, drei Spalten und vier Schichten umfassen soll. Ein dreidimensionales array lässt sich nämlich als Quader vorstellen, der aus mehreren zweidimensionalen Matrizen besteht, die in Schichten hintereinander gereiht sind. Das Argument `dimnames` dient dazu, die Dimensionen und die einzelnen Stufen in jeder Dimension gleichzeitig mit Namen zu versehen. Dies geschieht unter Verwendung einer Liste (vgl. Abschn. 3.1), die für jede Dimension eine Komponente in Form eines Vektors mit den gewünschten Bezeichnungen der einzelnen Stufen besitzt. Die Namen der Dimensionen selbst können über die Benennung der Komponenten der Liste festgelegt werden.

Als Beispiel soll die Kontingenztafel dreier kategorialer Variablen dienen: Geschlecht mit zwei und zwei weitere Variablen mit drei bzw. zwei Stufen. Ein dreidimensionales array wird durch separate zweidimensionale Matrizen für jede Stufe der dritten Dimension ausgegeben.

```
> (myArr1 <- array(1:12, c(2, 3, 2), dimnames=list(row=c("f", "m"),
+                                         column=c("CG", "WL", "T"), layer=c("high", "low"))))
, , layer = high
    column
row CG  WL  T
f   1   3   5
m   2   4   6

, , layer = low
    column
row CG  WL  T
f   7   9  11
m   8  10  12
```

Das array wird durch die mit dem Vektor `1:12` bereitgestellten Daten in Reihenfolge der Dimensionen aufgefüllt: zunächst alle Zeilen der ersten Spalte der ersten Schicht, dann in diesem Muster alle Spalten der ersten Schicht und zuletzt in diesem Muster alle Schichten. Auf arrays lassen sich mit `apply()` wie bei Matrizen beliebige Funktionen in Richtung der einzelnen Dimensionen anwenden.

Arrays lassen sich analog zu Matrizen mit dem `[<Index>]` Operator indizieren, wobei die Indizes für die zusätzlichen Dimensionen durch Komma getrennt hinzugefügt werden.

```
> myArr1[1, 3, 2]          # Element in 1. Zeile, 3. Spalte, 2. Schicht
[1] 11
```

³⁹So, wie sich Matrizen mit `cbind()` und `rbind()` aus Vektoren zusammenstellen lassen, ermöglicht `abind()` aus dem gleichnamigen Paket (Plate & Heiberger, 2011) das Verbinden von Matrizen zu einem array.

```
> myArr2 <- myArr1*2
> myArr2[ , , "high"]          # zeige nur 1. Schicht
   column
row  CG  WL  T
f    2   6  10
m    4   8  12
```

Ähnlich wie sich bei Matrizen durch Transponieren mit `t()` Zeilen und Spalten vertauschen lassen, können mit `aperm()` auch bei arrays Dimensionen ausgetauscht werden.

```
> aperm(a=<array>, perm=<Vektor>)
```

Unter `a` ist das zu transformierende n -dimensionale array anzugeben. `perm` legt in Form eines Vektors mit den Elementen 1 bis n fest, welche Dimensionen vertauscht werden sollen. Im Fall benannter Dimensionen kann `perm` alternativ auch deren Namen enthalten. Die Position eines Elements von `perm` bezieht sich auf die alte Dimension, das Element selbst bestimmt, zu welcher neuen Dimension die alte gemacht wird. Sollen in einem dreidimensionalen array die Schichten zu Zeilen (und umgekehrt) werden, wäre `perm=c(3, 2, 1)` zu setzen. Das Vertauschen von Zeilen und Spalten wäre mit `perm=c(2, 1, 3)` zu erreichen.

2.10 Häufigkeitsauszählungen

Bei der Analyse kategorialer Variablen besteht ein typischer Auswertungsschritt darin, die Auftretenshäufigkeiten der Kategorien auszuzählen und relative sowie bedingte relative Häufigkeiten zu berechnen. Wird nur eine Variable betrachtet, ergeben sich einfache Häufigkeitstabellen, bei mehreren Variablen mehrdimensionale Kontingenztafeln der gemeinsamen Häufigkeiten.

2.10.1 Einfache Tabellen absoluter und relativer Häufigkeiten

Eine Tabelle der absoluten Häufigkeiten von Variablenwerten erstellt `table(<Faktor>)` und erwartet dafür als Argument ein eindimensionales Objekt, das sich als Faktor interpretieren lässt, z. B. einen Vektor. Das Ergebnis ist eine Übersicht über die Auftretenshäufigkeit jeder vorkommenden Ausprägung, wobei fehlende Werte ignoriert werden.⁴⁰

```
> (myLetters <- sample(LETTERS[1:5], 12, replace=TRUE))
[1] "C" "D" "A" "D" "E" "D" "C" "E" "E" "B" "E" "E"

> (tab <- table(myLetters))
myLetters
A  B  C  D  E
1  1  2  3  5
```

⁴⁰Ist das erste Argument von `table()` ein Vektor, können fehlende Werte über das Argument `exclude=NULL` mit in die Auszählung einbezogen werden. Damit in Faktoren vorkommende fehlende Werte unter einer eigenen Kategorie berücksichtigt werden, muss der Faktor `NA` als eigene Stufe enthalten und deshalb mit `factor(<Vektor>, exclude=NULL)` gebildet werden.

In der oberen Zeile der Ausgabe sind die Ausprägungen der Variable, in der unteren Zeile die jeweils zugehörigen Auftretenshäufigkeiten aufgeführt. Eindimensionale Häufigkeitstabellen verhalten sich wie Vektoren mit benannten Elementen, wobei die Benennungen den vorhandenen Ausprägungen der Variable entsprechen. Die Ausprägungen lassen sich mit dem Befehl `names(<Tabelle>)` separat ausgeben.

```
> names(tab)
[1] "A" "B" "C" "D" "E"

> tab["B"]
B
1
```

Relative Häufigkeiten ergeben sich durch Division der absoluten Häufigkeiten mit der Gesamtzahl der Beobachtungen. Für diese Rechnung existiert die Funktion `prop.table(<Tabelle>)`, welche als Argument eine Tabelle der absoluten Häufigkeiten erwartet und die relativen Häufigkeiten ausgibt. Durch Anwendung von `cumsum()` auf das Ergebnis erhält man die kumulierten relativen Häufigkeiten (für eine andere Methode und die Berechnung von Prozenträngen vgl. Abschn. 2.10.6).

```
> (relFreq <- prop.table(table(myLetters)))
myLetters
      A          B          C          D          E 
0.08333333 0.08333333 0.16666667 0.25000000 0.41666667

> cumsum(relFreq)           # kumulierte relative Häufigkeiten
      A          B          C          D          E 
0.08333333 0.16666667 0.33333333 0.58333333 1.00000000
```

Kommen mögliche Variablenwerte in einem Vektor nicht vor, so tauchen sie auch in einer mit `table()` erstellten Häufigkeitstabelle nicht als ausgezählte Kategorie auf. Um deutlich zu machen, dass Variablen außer den tatsächlich vorhandenen Ausprägungen potentiell auch weitere Werte annehmen, kann auf zweierlei Weise vorgegangen werden: So können die möglichen, tatsächlich aber nicht auftretenden Werte der Häufigkeitstabelle nachträglich mit der Häufigkeit 0 hinzugefügt werden.

```
> tab["Q"] <- 0
> tab
  A  B  C  D  E  Q
  1  1  2  3  5  0
```

Beim Hinzufügen von Werten zur Tabelle werden diese ans Ende angefügt. Geht dadurch die natürliche Reihenfolge der Ausprägungen verloren, kann die Tabelle z. B. mittels eines durch `order(names(<Tabelle>))` erstellten Vektors der geordneten Indizes sortiert werden.

```
> tabOrder <- order(names(tab))
> tab[tabOrder]           # ...
```

Alternativ zur Veränderung der Tabelle selbst können die Daten vorab in einen Faktor umgewandelt werden. Dem Faktor lässt sich dann der nicht auftretende, aber prinzipiell mögliche Wert als weitere Stufe hinzufügen.

```
> letFac <- factor(myLetters, levels=c(LETTERS[1:5], "Q"))
> letFac
[1] C D A D E D C E E B E E
Levels: A B C D E Q

> table(letFac)
letFac
A   B   C   D   E   Q
1   1   2   3   5   0
```

2.10.2 Iterationen zählen

Unter einer *Iteration* innerhalb einer linearen Sequenz von Symbolen ist ein Abschnitt zu verstehen, der aus der ein- oder mehrfachen Wiederholung desselben Symbols besteht (*run*). Iterationen werden durch Iterationen eines anderen Symbols begrenzt, oder besitzen kein vorangehendes bzw. auf sie folgendes Symbol. Die Iterationen eines Vektors zählt die `rle(Vektor)` Funktion, deren Ergebnis eine Liste mit zwei Komponenten ist: Die erste Komponente `lengths` ist ein Vektor, der die jeweilige Länge jeder Iteration als Elemente besitzt. Die zweite Komponente `values` ist ein Vektor mit den Symbolen, um die es sich bei den Iterationen handelt (vgl. Abschn. 3.1).

```
> (vec <- rep(rep(c("f", "m"), 3), c(1, 3, 2, 4, 1, 2)))
[1] "f" "m" "m" "m" "f" "f" "m" "m" "m" "m" "f" "m" "m" "m"

> (res <- rle(vec))
Run Length Encoding
lengths: int [1:6] 1 3 2 4 1 2
values : chr [1:6] "f" "m" "f" "m" "f" "m"

> length(res$lengths)                      # zähle Anzahl der Iterationen
[1] 6
```

Aus der jeweiligen Länge und dem wiederholten Symbol jeder Iteration lässt sich die ursprüngliche Sequenz eindeutig rekonstruieren. Dies kann durch die `inverse.rle(rle-Ergebnis)` Funktion geschehen, die eine Liste erwartet, wie sie `rle()` als Ergebnis besitzt.

```
> inverse.rle(res)
[1] "f" "m" "m" "m" "f" "f" "m" "m" "m" "m" "f" "m" "m"
```

2.10.3 Absolute, relative und bedingte relative Häufigkeiten in Kreuztabellen

Statt die Häufigkeiten der Werte nur einer einzelnen Variable zu ermitteln, können mit `table(Faktor1, Faktor2, ...)` auch mehrdimensionale Kontingenztafeln erstellt werden.

Die Elemente der Faktoren an gleicher Position werden als denselben Beobachtungsobjekten zugehörig interpretiert. Das erste Element von `<Faktor1>` bezieht sich also auf dieselbe Beobachtung wie das erste Element von `<Faktor2>`, usw. Das Ergebnis ist eine Kreuztabelle mit den gemeinsamen absoluten Häufigkeiten der Merkmale, wobei die Ausprägungen des ersten Faktors in den Zeilen stehen.

Als Beispiel sollen Personen betrachtet werden, die nach ihrem Geschlecht und dem Ort ihres Arbeitsplatzes unterschieden werden. An diesen Personen sei weiterhin eine Variable erhoben worden, die die absolute Häufigkeit eines bestimmten Ereignisses codiert.

```
> N      <- 10
> (work <- factor(sample(c("home", "office"), N, replace=TRUE)))
[1] home office office home home office office office office
Levels: home office

> (sex <- factor(sample(c("f", "m"), N, replace=TRUE)))
[1] f m f m m f m f m m
Levels: f m

> (counts <- sample(0:5, N, replace=TRUE))
[1] 0 3 3 1 1 1 3 5 2 4

# gemeinsame absolute Häufigkeiten von Geschlecht und Arbeitsplatz
> (absFreq <- table(sex, work))
   work
sex   home   office
  f       1       3
  m       2       4
```

Um relative Häufigkeiten auf Basis von Kreuztabellen absoluter Häufigkeiten zu ermitteln, eignet sich die `prop.table()` Funktion. Für bedingte relative Häufigkeiten besitzt sie ein zweites Argument `margin=Nummer`, an das etwa 1 zur Bestimmung der auf die Zeilen bezogenen bedingten relativen Häufigkeiten übergeben werden kann. Jede Zeile der Tabelle der relativen Häufigkeiten wird dann durch die zugehörige Zeilensumme dividiert, was sich manuell auch durch `sweep()` erreichen ließe. Mit `margin=2` erhält man analog die auf die Spalten bezogenen bedingten relativen Häufigkeiten.

```
> (relFreq <- prop.table(absFreq))                      # relative Häufigkeiten
   work
sex   home   office
  f     0.1     0.3
  m     0.2     0.4

> prop.table(absFreq, 1)          # auf Zeilen bedingte rel. Häufigkeiten
   work
sex      home      office
  f  0.2500000  0.7500000
  m  0.3333333  0.6666667
```

```
> prop.table(absFreq, 2)          # auf Spalten bedingte rel. Häufigkeiten
                                work
sex      home      office
f    0.3333333  0.4285714
m    0.6666667  0.5714286

# manuelle Kontrolle
> rSums <- rowSums(relFreq)      # Zeilensummen
> cSums <- colSums(relFreq)      # Spaltensummen
> sweep(relFreq, 1, rSums, "/") # auf Zeilen bedingte rel. Häufigkeiten ...
> sweep(relFreq, 2, cSums, "/") # auf Spalten bedingte rel. Häufigkeiten ...
```

Um Häufigkeitsauszählungen für mehr als zwei Variablen zu berechnen, können beim Aufruf von `table()` einfach weitere Faktoren durch Komma getrennt hinzugefügt werden. Die Ausgabe verhält sich dann wie ein array. Hierbei werden etwa im Fall von drei Variablen so viele zweidimensionale Kreuztabelle ausgegeben, wie Stufen der dritten Variable vorhanden sind. Soll dagegen auch in diesem Fall eine einzelne Kreuztabelle mit verschachteltem Aufbau erzeugt werden, ist `ftable()` (*flat table*) zu nutzen.

```
> ftable(x, row.vars=NULL, col.vars=NULL)
```

Unter `x` kann entweder eine bereits mit `table()` erzeugte Kreuztabelle eingetragen werden, oder aber eine durch Komma getrennte Reihe von Faktoren bzw. von Objekten, die sich als Faktor interpretieren lassen. Die Argumente `row.vars` und `col.vars` kontrollieren, welche Variablen in den Zeilen und welche in den Spalten angeordnet werden. Beide Argumente akzeptieren numerische Vektoren mit den Nummern der entsprechenden Variablen, oder aber Vektoren aus Zeichenketten, die den Namen der Faktoren entsprechen.

```
> (group <- factor(sample(c("A", "B"), 10, replace=TRUE)))
[1] B B A B A B A B A A
Levels: A B
```

```
> ftable(work, sex, group, row.vars="work", col.vars=c("sex", "group"))
      sex      f      m
      group   A   B   A   B
work
home        0   1   1   1
office       1   2   3   1
```

Beim Erstellen von Kreuztabellen kann auch auf `xtabs()` zurückgegriffen werden, insbesondere wenn sich die Variablen in Datensätzen befinden (vgl. Abschn. 3.2).

```
> xtabs(formula=~ ., data=<Datensatz>)
```

Im ersten Argument `formula` erwartet `xtabs()` eine *Modellformel* (vgl. Abschn. 5.2). Hier ist dies eine besondere Art, die in der Kontingenztafel zu berücksichtigenden Faktoren aufzuzählen, die durch ein + verknüpft rechts von der Tilde ~ stehen. In der Voreinstellung ~ . werden alle

Variablen des unter `data` angegebenen Datensatzes einbezogen, d. h. alle möglichen Kombinationen von Faktorstufen gebildet. Eine links der `~` genannte Variable wird als Vektor von Häufigkeiten interpretiert, die pro Stufenkombination der rechts von der `~` genannten Faktoren zu addieren sind. Stammen die in der Modellformel genannten Variablen aus einem Datensatz, muss dieser unter `data` aufgeführt werden. Die Ausprägungen des rechts der `~` zuerst genannten Faktors bilden die Zeilen der ausgegebenen Kontingenztafel.

```
> (persons <- data.frame(sex, work, counts))      # Variablen als Datensatz
   sex   work  counts
1   f    home      0
2   m  office      3
3   f  office      3
4   m    home      1
5   m    home      1
6   f  office      1
7   m  office      3
8   f  office      5
9   m  office      2
10  m  office      4

# gemeinsame Häufigkeiten der Stufen von sex und work
> xtabs(~ sex + work, data=persons)
      work
sex  home  office
  f     1      3
  m     2      4

# Summe von counts pro Zelle
> xtabs(counts ~ sex + work, data=persons)
      work
sex  home  office
  f     0      9
  m     2     12
```

Einen Überblick über die Zahl der in einer Häufigkeitstabelle ausgewerteten Faktoren sowie die Anzahl der zugrundeliegenden Beobachtungsobjekte erhält man mit `summary(⟨Tabelle⟩)`. Im Fall von Kreuztabellen wird hierbei zusätzlich ein χ^2 -Test auf Unabhängigkeit bzw. auf Gleichheit von Verteilungen berechnet (vgl. Abschn. 10.2.1, 10.2.2).

```
> summary(table(sex, work))
Number of cases in table: 10
Number of factors: 2
Test for independence of all factors:
Chisq = 4.444, df = 1, p-value = 0.03501
Chi-squared approximation may be incorrect
```

2.10.4 Randkennwerte von Kreuztabellen

Um Randsummen, Randmittelwerte oder ähnliche Kennwerte für eine Kreuztabelle zu berechnen, können alle für Matrizen vorgestellten Funktionen verwendet werden, insbesondere `apply()`, aber etwa auch `rowSums()` und `colSums()` sowie `rowMeans()` und `colMeans()`. Hier nimmt die Tabelle die Rolle der Matrix ein.

```
# Zeilensummen
> apply(xtabs(~ sex + work, data=persons), MARGIN=1, FUN=sum)
f   m
4   6

# Spaltenmittel
> colMeans(xtabs(~ sex + work, data=persons))
home office
1.5     3.5
```

`addmargins()` berechnet beliebige Randkennwerte für eine Kreuztabelle `A` entsprechend der mit dem Argument `FUN` bezeichneten Funktion (Voreinstellung ist `sum` für Randsummen). Die Funktion operiert separat über jeder der mit dem Vektor `margin` bezeichneten Dimensionen. Die Ergebnisse der Anwendung von `FUN` werden `A` in der Ausgabe als weitere Zeile und Spalte hinzugefügt.

```
> addmargins(A=<Tabelle>, margin=<Vektor>, FUN=<Funktion>)

# Randmittel
> addmargins(xtabs(~ sex + work, data=persons), c(1, 2), FUN=mean)
      work
sex  home  office  mean
  f    1.0    3.0    2.0
  m    2.0    4.0    3.0
mean 1.5    3.5    2.5
```

2.10.5 Datensätze aus Häufigkeitstabellen erstellen

In manchen Situationen liegen Daten nur in Form von Häufigkeitstabellen vor – etwa wenn sie aus der Literatur übernommen werden. Um die Daten selbst auszuwerten, ist es dann notwendig, sie zunächst in ein Format umzuwandeln, das die Variablen als separate Spalten verwendet und pro Beobachtungsobjekt eine Zeile besitzt. Diese Aufgabe erledigt die Funktion `Untable()` aus dem Paket `DescTools`. Als Ausgabe liefert sie einen Datensatz (vgl. Abschn. 3.2).

```
> cTab <- xtabs(~ sex + work, data=persons)      # Kreuztabelle
> library(DescTools)                                # für Untable()
> Untable(cTab)
      sex   work
1   f   home
2   f office
```

```

3   f office
4   f office
5   m  home
6   m  home
7   m office
8   m office
9   m office
10  m office

```

Eine andere Darstellung liefert die explizite Umwandlung der Kreuztabelle in einen Datensatz mit `as.data.frame()`. Der erzeugte Datensatz besitzt für jede Zelle der Kreuztabelle eine Zeile und neben den Spalten für die ausgezählten Variablen eine weitere Spalte `Freq` mit der Häufigkeit der Merkmalskombination.

```

> as.data.frame(cTab)
  sex   work Freq
1  f    home   1
2  m    home   2
3  f  office   3
4  m  office   4

```

2.10.6 Kumulierte relative Häufigkeiten und Prozentrang

`ecdf(x=<Vektor>)` (*empirical cumulative distribution function*) ermittelt die kumulierten relativen Häufigkeiten kategorialer Daten. Diese geben für einen Wert x_i an, welcher Anteil der Daten nicht größer als x_i ist. Das Ergebnis ist analog zur Verteilungsfunktion quantitativer Zufallsvariablen.

Das Ergebnis von `ecdf()` ist eine Stufenfunktion mit so vielen Sprungstellen, wie es unterschiedliche Werte in `x` gibt. Die Höhe jedes Sprungs entspricht der relativen Häufigkeit des Wertes an der Sprungstelle. Enthält `x` also keine mehrfach vorkommenden Werte, erzeugt `ecdf()` eine Stufenfunktion mit so vielen Sprungstellen, wie `x` Elemente besitzt. Dabei weist jeder Sprung dieselbe Höhe auf – die relative Häufigkeit `1/length(x)` jedes Elements von `x`. Tauchen in `x` Werte mehrfach auf, unterscheiden sich die Sprunghöhen dagegen entsprechend den relativen Häufigkeiten.

Die Ausgabe von `ecdf()` ist ihrerseits eine Funktion, die zunächst einem eigenen Objekt zugewiesen werden muss, ehe sie zur Ermittlung kumulierter relativer Häufigkeiten Verwendung finden kann. Ist `Fn()` diese Stufenfunktion, und möchte man die kumulierten relativen Häufigkeiten der in `x` gespeicherten Werte erhalten, ist `x` selbst als Argument für `Fn()` einzusetzen. Andere Werte als Argument von `Fn()` sind aber ebenfalls möglich. Indem `Fn()` für beliebige Werte ausgewertet wird, lassen sich empirische und interpolierte Prozentränge ermitteln. Mit `ecdf()` erstellte Funktionen können über `plot()` in einem Diagramm gezeigt werden (Abb. 10.1, Abb. 14.22, vgl. Abschn. 14.6.6).

```

> (vec <- round(rnorm(10), 2))
[1] -1.57 2.21 -1.01 0.21 -0.29 -0.61 -0.17 1.90 0.17 0.55

```

```
# kumulierte relative Häufigkeiten der vorhandenen Werte
> Fn <- ecdf(vec)
> Fn(vec)
[1] 0.1 1.0 0.2 0.7 0.4 0.3 0.5 0.9 0.6 0.8

> 100 * Fn(0.1)                                # Prozentrang von 0.1
[1] 50

> 100 * (sum(vec <= 0.1) / length(vec))        # Kontrolle
[1] 50
```

Soll die Ausgabe der kumulierten relativen Häufigkeiten in der richtigen Reihenfolge erfolgen, müssen die Werte mit `sort()` geordnet werden.

```
> Fn(sort(vec))
[1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Die Sprungstellen einer von `ecdf()` erstellten Funktion lassen sich mit `knots()` extrahieren. Das Ergebnis sind gerade die in `x` enthaltenen unterschiedlichen sortierten Werte.

```
> knots(Fn)
[1] -1.57 -1.01 -0.61 -0.29 -0.17 0.17 0.21 0.55 1.90 2.21
```

2.11 Fehlende Werte behandeln

Empirische Datensätze besitzen häufig zunächst keine zufriedenstellende Qualität, etwa durch Fehler in der Eingabe von Werten, Ausreißer (vgl. Abschn. 6.5.1, 14.6) oder durch unvollständige Daten – wenn also nicht für alle Beobachtungsobjekte Werte von allen erhobenen Variablen vorliegen. So können etwa Personen die Auskunft bzgl. bestimmter Fragen verweigern, Aufgaben übersehen oder in adaptiven Tests aufgrund von Verzweigungen nicht vorgelegt bekommen.

Fehlende Werte bergen aus versuchsplanerischer Perspektive die Gefahr, dass sie womöglich nicht zufällig, sondern systematisch entstanden sind und so zu verzerrten Ergebnissen führen. Aber auch bei der statistischen Auswertung werfen sie Fragen auf: Zum einen sind verschiedene Strategien des Umgangs mit ihnen denkbar, die nicht unbedingt zu gleichen Ergebnissen führen. Zum anderen können fehlende Werte bewirken, dass nicht wie beabsichtigt in allen Experimentalbedingungen dieselbe Anzahl von Beobachtungen vorliegt. Dies ist jedoch für die Anwendung und Interpretation vieler üblicher Verfahren relevant, deren Einsatz durch das Vorliegen fehlender Werte problematisch werden könnte.

Um fehlende Werte in Indexvektoren zu vermeiden, wo sie meist zu nicht intendierten Konsequenzen führen, sollten logische Indexvektoren mit `which()` in numerische konvertiert werden (vgl. Abschn. 2.2.2).

2.11.1 Fehlende Werte codieren und identifizieren

Wenn ein Datensatz eingegeben wird und fehlende Werte vorliegen, so dürfen diese nicht einfach weggelassen, sondern müssen mit der Konstante NA (*not available*) codiert werden – auch bei character Vektoren ist sie nicht in Anführungszeichen zu setzen.⁴¹

```
> (vec1 <- c(10, 20, NA, 40, 50, NA))
[1] 10 20 NA 40 50 NA
```

```
> length(vec1)
[1] 6
```

In manchen Situationen werden Zeichenketten prinzipiell ohne Anführungszeichen ausgegeben, etwa bei Faktoren oder Datensätzen (vgl. Abschn. 3.2). Zur leichteren Unterscheidung von gültigen Zeichenketten erscheinen fehlende Werte deshalb dann als <NA>.⁴²

```
# Ausgabe von Zeichenketten mit Anführungszeichen -> nur NA
> LETTERS[c(1, NA, 3)]
[1] "A" NA "C"
```

```
# Ausgabe von Zeichenketten ohne Anführungszeichen -> <NA>
> factor(LETTERS[c(1, NA, 3)])
[1] A <NA> C
Levels: A C
```

Ob in einem Vektor fehlende Werte vorhanden sind, wird mit der Funktion `is.na(Vektor)` ermittelt.⁴³ Sie gibt einen logischen Vektor aus, der für jede Position angibt, ob das Element ein fehlender Wert ist. Im Fall einer Datenmatrix liefert `is.na()` eine Matrix aus Wahrheitswerten, die für jedes Matrixelement angibt, ob es sich um einen fehlenden Wert handelt.

```
> is.na(vec1)
[1] FALSE FALSE TRUE FALSE FALSE TRUE
```

```
> vec2 <- c(NA, 7, 9, 10, 1, 8)
> (matNA <- rbind(vec1, vec2))
     [,1] [,2] [,3] [,4] [,5] [,6]
vec1   10    20   NA    40    50   NA
vec2   NA     7     9    10     1     8
```

```
> is.na(matNA)
```

⁴¹Für jeden Datentyp existiert jeweils eine passende Konstante, all diese Konstanten tragen aber den Namen NA. Der mit `typeof(NA)` ausgegebene Basis-Datentyp ist `logical`. Damit ist die Ausgabe etwa von `c(1, 2, 3, NA)` gleich `NA NA NA`, da logische Indexvektoren zyklisch verlängert werden (vgl. Abschn. 2.1.2, 2.2.2).

⁴²In solchen Situationen ist also NA die Ausgabe eines gültigen Elements und von einem fehlenden Wert <NA> zu unterscheiden. So erzeugt `factor(c("A", "NA", "C"))[c(NA, 2, 3)]` die Ausgabe <NA> NA C. Einzig die gültige Zeichenkette "<NA>" lässt sich in der Ausgabe nicht von einem fehlenden Wert unterscheiden. In diesem Fall kann nur mit Hilfe von `is.na()` festgestellt werden, ob es sich tatsächlich um einen fehlenden Wert handelt.

⁴³Der == Operator eignet sich nicht zur Prüfung auf fehlende Werte, da das Ergebnis von `Wert == NA` selbst NA ist (vgl. Abschn. 2.11.3).

```
[,1] [,2] [,3] [,4] [,5] [,6]
vec1 FALSE FALSE TRUE FALSE FALSE TRUE
vec2 TRUE FALSE FALSE FALSE FALSE FALSE
```

Bei einem großen Datensatz ist es mühselig, die Ausgabe von `is.na()` manuell nach TRUE Werten zu durchsuchen. Daher bietet sich `any()` an, um zu erfahren, ob überhaupt fehlende Werte vorliegen, `sum()`, um deren Anzahl und `which()`, um deren Position zu ermitteln.

```
> any(is.na(vec1)) # gibt es fehlende Werte?
[1] TRUE

> sum(is.na(vec1)) # wie viele?
[1] 2

> which(is.na(vec1)) # wo im Vektor?
[1] 3 6

> which(is.na(matNA), arr.ind=TRUE) # wo in der Matrix?
  row col
vec2  2   1
vec1  1   3
vec1  1   6
```

2.11.2 Fehlende Werte ersetzen und umcodieren

Fehlende Werte werden bei der Dateneingabe in anderen Programmen oft mit Zahlen codiert, die keine mögliche Ausprägung einer Variable sind, z.B. mit 999. Bisweilen ist diese Codierung auch nicht einheitlich, sondern verwendet verschiedene Zahlen, etwa wenn Daten aus unterschiedlichen Quellen zusammengeführt werden. Bei der Verarbeitung von aus anderen Programmen übernommenen Datensätzen in R muss die Codierung fehlender Werte also ggf. angepasst werden (vgl. Abschn. 4.2, insbesondere das Argument `na.strings` in Abschn. 4.2.5 und 4.2.2).

Die Identifikation der zu ersetzenden Werte kann über `<Vektor> %in% <Menge>` erfolgen, wobei `<Menge>` ein Vektor mit allen Werten ist, die als fehlend gelten sollen (vgl. Abschn. 2.3.2). Der damit erzeugte Indexvektor lässt sich direkt an das Ergebnis von `is.na()` zuweisen, wodurch die zugehörigen Elemente auf NA gesetzt werden. Das Vorgehen bei Matrizen ist analog.

```
# fehlende Werte sind zunächst mit -999 und 999 codiert
> vec <- c(30, 25, 23, 21, -999, 999) # Vektor mit fehlenden Werten
> is.na(vec) <- vec %in% c(-999, 999) # ersetze missings durch NA
> vec
[1] 30 25 23 21 NA NA

# Matrix mit fehlenden Werten
> (mat <- matrix(c(30, 25, 23, 21, -999, 999), nrow=2, ncol=3))
 [,1] [,2] [,3]
```

```
[1,] 30 23 -999
[2,] 25 21 999

> is.na(mat) <- mat %in% c(-999, 999) # ersetze missings durch NA
> mat
     [,1] [,2] [,3]
[1,] 30   23   NA
[2,] 25   21   NA
```

2.11.3 Behandlung fehlender Werte bei der Berechnung einfacher Kennwerte

Wenn in einem Vektor oder einer Matrix fehlende Werte vorhanden sind, muss Funktionen zur Berechnung statistischer Kennwerte über ein Argument angegeben werden, wie mit ihnen zu verfahren ist. Andernfalls kann der Kennwert nicht berechnet werden, und das Ergebnis der Funktion ist seinerseits `NA`.⁴⁴ Allerdings lassen sich `NA` Einträge zunächst manuell entfernen, ehe die Daten an eine Funktion übergeben werden. Zu diesem Zweck existiert die Funktion `na.omit(<Vektor>)`, die das übergebene Objekt um fehlende Werte bereinigt ausgibt.

```
> sd(na.omit(vecNA))          # um NA bereinigten Vektor übergeben
[1] 5.125102

# fehlende Werte manuell entfernen
> goodIdx <- !is.na(vecNA)      # Indizes der nicht fehlenden Werte
> mean(vecNA[goodIdx])         # um NA bereinigten Vektor übergeben
[1] 24.33333
```

Die Behandlung fehlender Werte lässt sich in vielen Funktionen auch direkt über das Argument `na.rm` steuern. In der Voreinstellung `FALSE` sorgt es dafür, dass fehlende Werte nicht stillschweigend bei der Berechnung des Kennwertes ausgelassen werden, sondern das Ergebnis `NA` ist. Soll der Kennwert dagegen auf Basis der vorhandenen Werte berechnet werden, muss das Argument `na.rm=TRUE` gesetzt werden.

```
> sum(vecNA)
[1] NA

> sum(vecNA, na.rm=TRUE)
[1] 146

> apply(matNA, 1, FUN=mean)
[1] NA NA

> apply(matNA, 1, mean, FUN=na.rm=TRUE)
[1] 23.33333 25.33333
```

⁴⁴ Allgemein ist das Ergebnis aller Rechnungen `NA`, sofern der fehlende Wert für das Ergebnis relevant ist. Ist das Ergebnis auch ohne den fehlenden Wert eindeutig bestimmt, wird es ausgegeben – so erzeugt `TRUE | NA` die Ausgabe `TRUE`, da sich bei einem logischen ODER das zweite Argument nicht auf das Ergebnis auswirkt, wenn das erste `WAHR` ist.

Auf die dargestellte Weise lassen sich fehlende Werte u.a. in `sum()`, `prod()`, `range()`, `mean()`, `median()`, `quantile()`, `var()`, `sd()`, `cov()` und `cor()` behandeln.

2.11.4 Behandlung fehlender Werte in Matrizen

Bei `cov(<Vektor1>, <Vektor2>)` und `cor(<Vektor1>, <Vektor2>)` bewirkt das `na.rm=TRUE` Argument, dass ein aus zugehörigen Werten von `<Vektor1>` und `<Vektor2>` gebildetes Wertepaar nicht in die Berechnung von Kovarianz oder Korrelation eingeht, wenn wenigstens einer der beiden Werte NA ist.

Zur Behandlung fehlender Werte stehen bei `cov()` und `cor()` außer dem Argument `na.rm=TRUE` mit dem *zeilenweisen* bzw. *paarweisen* Fallausschluss weitere Möglichkeiten zur Verfügung. Sie sind aber erst relevant, wenn Kovarianz- bzw. Korrelationsmatrizen auf Basis von Matrizen mit mehr als zwei Spalten erstellt werden.

Zeilenweiser (fallweiser) Fallausschluss

Beim zeilenweisen Fallausschluss werden die Matrixzeilen komplett entfernt, in denen NA Werte auftauchen, ehe die Matrix für Berechnungen herangezogen wird. Weil eine Zeile oft allen an einem Beobachtungsobjekt erhobenen Daten entspricht, wird dies auch als fallweiser Fallausschluss bezeichnet. `na.omit(<Matrix>)` bereinigt eine Matrix mit fehlenden Werten um Zeilen, in denen NA Einträge auftauchen. Die Indizes der dabei ausgeschlossenen Zeilen werden der ausgegebenen Matrix als Attribut hinzugefügt. Mit der so gebildeten Matrix fließen im Beispiel die Zeilen 1 und 2 nicht mit in Berechnungen ein.

```

> ageNA  <- c(18, NA, 27, 22)
> DV1    <- c(NA, 1, 5, -3)
> DV2    <- c(9, 4, 2, 7)
> (matNA <- cbind(ageNA, DV1, DV2))
      ageNA   DV1   DV2
[1,]     18    NA     9
[2,]     NA     1     4
[3,]     27     5     2
[4,]     22    -3     7

> na.omit(matNA)                                # Zeilen mit NA entfernen
      ageNA   DV1   DV2
[1,]     27     5     2
[2,]     22    -3     7

attr("na.action")
[1] 2 1

attr("class")
[1] "omit"

```

```
> colMeans(na.omit(matNA)) # Berechnung ohne NAs
ageNA DV1 DV2
24.5 1.0 4.5
```

Alternativ besteht die Möglichkeit, alle Zeilen mit fehlenden Werten anhand des Befehls `apply(is.na(Matrix), 1, any)` selbst festzustellen. Mit dem resultierenden logischen Indexvektor lassen sich die betroffenen Zeilen von weiteren Berechnungen ausschließen.

```
> (rowNAidx <- apply(is.na(matNA), 1, any)) # Zeilen mit NA feststellen
[1] TRUE TRUE FALSE FALSE

> matNA[!rowNAidx, ] # Zeilen mit NA entfernen
ageNA DV1 DV2
[1,] 27 5 2
[2,] 22 -3 7
```

Bei `cov()` und `cor()` bewirkt bei der Berechnung von Kovarianz- und Korrelationsmatrizen mit mehr als zwei Variablen das Argument `use="complete.obs"` den fallweisen Ausschluss fehlender Werte. Dessen Verwendung hat denselben Effekt wie die vorherige Reduktion der Matrix um Zeilen, in denen fehlende Werte auftauchen.

```
> cov(matNA, use="complete.obs")
age DV1 DV2
age 12.5 20 -12.5
DV1 20.0 32 -20.0
DV2 -12.5 -20 12.5

# beide Arten des fallweisen Ausschlusses erzielen dasselbe Ergebnis
> all.equal(cov(matNA, use="complete.obs"), cov(na.omit(matNA)))
[1] TRUE
```

Paarweiser Fallausschluss

Beim paarweisen Fallausschluss werden die Werte einer auch `NA` beinhaltenden Zeile soweit als möglich in Berechnungen berücksichtigt, die Zeile wird also nicht vollständig ausgeschlossen. Welche Werte einer Zeile Verwendung finden, hängt von der konkreten Auswertung ab. Der paarweise Fallausschluss wird im Fall der Berechnung der Summe oder des Mittelwertes über Zeilen oder Spalten mit dem Argument `na.rm=TRUE` realisiert, das alle Werte außer `NA` einfließen lässt.

```
> rowMeans(matNA)
[1] NA NA 11.333333 8.666667

> rowMeans(mat, na.rm=TRUE)
[1] 13.500000 2.500000 11.333333 8.666667
```

Bei der Berechnung von Kovarianz- und Korrelationsmatrizen für mehr als zwei Variablen mit `cov()` und `cor()` bewirkt das Argument `use="pairwise.complete.obs"` den paarweisen

Ausschluss fehlender Werte. Es wird dann bei der Berechnung jeder Kovarianz pro Zeile geprüft, ob in den zugehörigen beiden Spalten ein gültiges Wertepaar existiert und dieses ggf. verwendet. Anders als beim fallweisen Ausschluss geschieht dies also auch dann, wenn in derselben Zeile Werte anderer Variablen fehlen, dies für die zu berechnende Kovarianz aber irrelevant ist.⁴⁵

Angewendet auf die Daten in `matNA` bedeutet das beispielsweise, dass beim fallweisen Ausschluss das von Beobachtungsobjekt 1 gelieferte Wertepaar nicht in die Berechnung der Kovarianz von `ageNA` und `DV2` einfließt, weil der Wert für `DV1` bei diesem Beobachtungsobjekt fehlt. Beim paarweisen Ausschluss werden diese Werte dagegen berücksichtigt. Lediglich bei der Berechnung der Kovarianz von `DV1` und `DV2` werden keine Daten des ersten Beobachtungsobjekts verwendet, weil ein Wert für `DV1` von ihr fehlt.

```
> cov(matNA, use="pairwise.complete.obs")
      ageNA   DV1       DV2
ageNA  20.33333  20  -16.000000
      DV1  20.00000  16  -10.000000
      DV2 -16.00000 -10   9.666667
```

Ob fehlende Werte fall- oder paarweise ausgeschlossen werden sollten, hängt u. a. von den Ursachen ab, warum manche Untersuchungseinheiten unvollständige Daten geliefert haben und andere nicht. Insbesondere stellt sich die Frage, ob Untersuchungseinheiten mit fehlenden Werten systematisch andere Eigenschaften haben, so dass von ihren Daten generell kein Gebrauch gemacht werden sollte.

2.11.5 Behandlung fehlender Werte beim Sortieren von Daten

Beim Sortieren von Daten mit `sort()` und `order()` wird die Behandlung fehlender Werte mit dem Argument `na.last` kontrolliert, das auf `NA`, `TRUE` oder `FALSE` gesetzt werden kann. Bei `sort()` ist `na.last` per Voreinstellung auf `NA` gesetzt und sorgt so dafür, dass fehlende Werte entfernt werden. Bei `order()` ist die Voreinstellung `TRUE`, wodurch fehlende Werte ans Ende plaziert werden. Auf `FALSE` gesetzt bewirkt `na.last` die Plazierung fehlender Werte am Anfang.

2.11.6 Behandlung fehlender Werte in inferenzstatistischen Tests

Viele Funktionen zur Berechnung statistischer Tests besitzen das Argument `na.action`, das festlegt, wie mit fehlenden Werten zu verfahren ist. Mögliche Werte sind u. a. die Namen der Funktionen `na.omit()` und `na.fail()`, die sich auch direkt auf Daten anwenden lassen (vgl. Abschn. 2.11.3, 3.3.4). Die Voreinstellung `na.omit` bewirkt den fallweisen Ausschluss (vgl. Abschn. 2.11.4), mit `na.fail` wird die Auswertung bei fehlenden Werten abgebrochen und eine Fehlermeldung ausgegeben. Global kann dieses Verhalten mit `options(na.action = "Wert")` geändert werden (vgl. `?na.action`). Generell empfiehlt es sich, Daten außerhalb von Auswertungsfunktionen um fehlende Werte zu bereinigen, bzw. sie mit Techniken der multiplen

⁴⁵Eine mit diesem Verfahren ermittelte Matrix kann auch nicht positiv semidefinit sein, und stellt dann keine Kovarianzmatrix bzw. Korrelationsmatrix im engeren Sinne dar.

Imputation zu vervollständigen. So lässt sich die Konsistenz bzgl. der in einzelne Auswertungen eingeflossenen Fälle besser sichern.

2.11.7 Multiple Imputation

Neben den beschriebenen Methoden zur Behandlung fehlender Werte versucht auch die *multiple Imputation* mit diesem Problem umzugehen. Sie ersetzt fehlende Werte eines aus mehreren Variablen bestehenden Datensatzes in mehreren Durchgängen jeweils durch solche Zahlen, die mit bestimmten Annahmen und unter Berücksichtigung der tatsächlich vorhandenen Daten generiert wurden. Die simulierten Daten sollen so die Verteilungseigenschaften der empirischen Daten teilen. Für jede einzelne Imputation aller fehlenden Werte lassen sich dann z. B. Parameterschätzungen einer linearen Regression berechnen. Die sich ergebene Menge unterschiedlicher Schätzungen – eine pro Imputation – kann dann geeignet zu einer Gesamtschätzung kombiniert werden.

Multiple Imputation wird in R u. a. von den Paketen **Amelia II** (Honaker, King & Blackwell, 2011), **mice** (van Buuren & Groothuis-Oudshoorn, 2011; van Buuren, 2012) und **mi** (Su, Gelman, Hill & Yajima, 2011) unterstützt. Für weitere vgl. den Abschnitt *Official Statistics & Survey Methodology* der CRAN Task Views (Templ, 2014).

2.12 Zeichenketten verarbeiten

Zeichenketten tauchen bei der eigentlichen Auswertung von Daten als Bezeichnungen für Variablen oder Gruppen auf. Vor allem bei der Aufbereitung eines Rohdatensatzes ist es aber hilfreich, sie flexibel erstellen und manipulieren zu können.⁴⁶

2.12.1 Objekte in Zeichenketten umwandeln

Mit `toString(<Objekt>)` lassen sich Objekte in Zeichenketten umwandeln. Das Ergebnis ist eine einzelne Zeichenkette mit den Inhalten des Objekts, wobei einzelne Elemente durch Komma mit folgendem Leerzeichen getrennt werden. Komplexe Objekte (z. B. Matrizen) werden dabei wie Vektoren verarbeitet.

```
> randVals <- round(rnorm(5), 2)
> toString(randVals)
[1] "-0.03, 1.01, -0.52, -1.03, 0.18"
```

Analog wandelt `capture.output(<Ausdruck>)` die normalerweise auf der Konsole erscheinende Ausgabe eines Befehlsausdrucks in eine Zeichenkette um. Die Ausgabe mehrerer Bearbeitungsschritte lässt sich mit `sink()` in eine Zeichenkette oder Datei leiten. Soll ein Protokoll aller Vorgänge als Kopie in eine Datei geschrieben werden, ist dabei das Argument `split=TRUE` zu setzen.

⁴⁶Das Paket **stringr** (Wickham, 2012) stellt für viele der im folgenden aufgeführten Funktionen Alternativen bereit, die den Umgang mit Zeichenketten erleichtern und konsistenter gestalten sollen.

`formatC()` ist auf die Umwandlung von Zahlen in Zeichenketten spezialisiert und bietet sich vor allem für die formatierte Ausgabe von Dezimalzahlen an.

```
> formatC(x=<Zahl>, digits=<Dezimalstellen>, width=<Breite>,
+           flag=<Modifikation>, format=<Zahlentyp>)"
```

Ist `x` eine Dezimalzahl, wird sie mit `digits` vielen Dezimalstellen ausgegeben. Die Angabe von `digits` fügt ganzen Zahlen keine Dezimalstellen hinzu, allerdings verbreitert sich die ausgegebene Zeichenkette auf `digits` viele Zeichen, indem `x` entsprechend viele Leerzeichen vorangestellt werden. Sollen der Zahl stattdessen Nullen vorangestellt werden, ist `flag="0"` zu setzen. Linksbündig ausgerichtete Zeichenketten sind mit `flag="-"` zu erreichen. Die Länge der Zeichenkette lässt sich auch unabhängig von der Zahl der Dezimalstellen mit dem Argument `width` kontrollieren. Schließlich ermöglicht `format` die Angabe, was für ein Zahlentyp bei `x` vorliegt, insbesondere ob es eine ganze Zahl ("d") oder eine Dezimalzahl ist. Im letztgenannten Fall kann die Ausgabeform etwa mit "f" wie gewohnt erfolgen (z. B. "1.234") oder mit "e" in wissenschaftlicher Notation (z. B. "1.23e+03") – für weitere Möglichkeiten vgl. `?formatC`.

```
> formatC(3, digits=5, format="d")
[1] "     3"

> formatC(c(1, 2.345), width=5, format="f")
[1] "1.0000" "2.3450"
```

2.12.2 Zeichenketten erstellen und ausgeben

Die einfachste Möglichkeit zum Erstellen eigener Zeichenketten ist ihre manuelle Eingabe auf der Konsole oder im Editor. Für Vektoren von Zeichenketten ist dabei zu beachten, dass der `length()` Befehl jede Zeichenkette als ein Element betrachtet. Dagegen gibt `nchar("<Zeichenkette>")` die Wortlänge jedes Elements an, aus wie vielen einzelnen Zeichen jede Zeichenkette des Vektors also besteht.

```
> length("ABCDEF")
[1] 1

> nchar("ABCDEF")
[1] 6

> nchar(c("A", "BC", "DEF"))
[1] 1 2 3
```

Fehlende Werte `NA` wandelt `nchar()` intern in die Zeichenkette "`NA`" um. Daher liefert `nchar(NA)` das Ergebnis 2.

Zeichenketten nach Muster erstellen

Die Methode, Zeichenketten manuell in Vektoren zu erstellen, stößt dort schnell an ihre Grenzen, wo sie von Berechnungen abhängen sollen oder viele Zeichenketten nach demselben Muster erzeugt werden müssen. `paste()` und `sprintf()` sind hier geeignete Alternativen.

Mit `paste()` lassen sich Zeichenketten mit einem bestimmten Aufbau erzeugen, indem verschiedene Komponenten aneinandergehängt werden, die etwa aus einem gemeinsamen Präfix und unterschiedlicher laufender Nummer bestehen können.

```
> paste(<Objekt1>, <Objekt2>, ..., sep=" ", collapse=NULL)
```

Die ersten Argumente von `paste()` sind Objekte, deren Elemente jeweils die Bestandteile der zu erstellenden Zeichenketten ausmachen und zu diesem Zweck aneinandergefügt werden. Das erste Element des ersten Objekts wird dazu mit den ersten Elementen der weiteren Objekte verbunden, ebenso die jeweils zweiten und folgenden Elemente. Das Argument `sep` kontrolliert, welche Zeichen jeweils zwischen den Elementen aufeinander folgender Objekte einzufügen sind – in der Voreinstellung ist dies das Leerzeichen. In der Voreinstellung `collapse=NULL` ist das Ergebnis ein Vektor aus Zeichenketten, wobei jedes seiner Elemente aus der Kombination jeweils eines Elements aus jedem übergebenen Objekt besteht. Hierbei werden unterschiedlich lange Vektoren ggf. zyklisch verlängert (vgl. Abschn. 2.5.4). Wird stattdessen für `collapse` eine Zeichenfolge übergeben, ist das Ergebnis eine einzelne Zeichenkette, deren Bestandteile durch diese Zeichenfolge getrennt sind.

```
> paste("group", LETTERS[1:5], sep="_")
[1] "group_A" "group_B" "group_C" "group_D" "group_E"
```

```
# Farben der Default-Farbspalte
> paste(1:5, palette()[1:5], sep=":")
[1] "1: black" "2: red" "3: green3" "4: blue" "5: cyan"
```

```
> paste(1:5, letters[1:5], sep=". ", collapse=" ")
[1] "1.a 2.b 3.c 4.d 5.e"
```

Fehlende Werte `NA` wandelt `paste()` in die Zeichenkette "`NA`" um. Treten innerhalb von `paste()` die leere Menge `NULL` oder leere Vektoren `character(0)` gemeinsam mit anderen Zeichen auf, werden sie wie die leere Zeichenkette der Länge 1 `""` behandelt, was an den eingefügten Trennzeichen `sep` deutlich wird.

```
> paste(1, NA, 2, NULL, 3, character(0), sep="_")
[1] "1_NA_2__3_"
```

Die an die gleichnamige Funktion der Programmiersprache C angelehnte Funktion `sprintf()` erzeugt Zeichenketten, deren Aufbau durch zwei Komponenten bestimmt wird: Einerseits durch einen die Formatierung und feste Elemente definierenden Teil (den *format string*), andererseits durch eine Reihe von Objekten, deren Werte an festgelegten Stellen des *format strings* einzufügen sind.

```
> sprintf(fmt="format string", <Objekt1>, <Objekt2>, ...)
```

Das Argument `fmt` erwartet eine Zeichenkette aus festen und variablen Elementen. Gewöhnliche Zeichen werden als feste Elemente interpretiert und tauchen unverändert in der erzeugten Zeichenkette auf. Variable Elemente werden durch das Prozentzeichen `%` eingeleitet, auf das ein Buchstabe folgen muss, der die Art des hier einzufügenden Wertes definiert. So gibt etwa `%d` an, dass hier ein ganzzahliger Wert einzufügen ist, `%f` dagegen weist auf eine Dezimalzahl hin

und `%s` auf eine Zeichenfolge. Das Prozentzeichen selbst wird durch `%%` ausgegeben, doppelte Anführungszeichen durch `\"`,⁴⁷ Tabulatoren durch `\t` und Zeilenumbrüche durch `\n` (vgl. ?Quotes).

Für jedes durch ein Prozentzeichen definierte Feld muss nach `fmt` ein passendes Objekt genannt werden, dessen Wert an der durch `%` bezeichneten Stelle eingefügt wird. Die Entsprechung zwischen variablen Feldern und Objekten wird über deren Reihenfolge hergestellt, der Wert des ersten Objekts wird also an der Stelle des ersten variablen Elements eingefügt, etc.

```
> N    <- 20
> grp <- "A"
> M    <- 14.2
> sprintf("For %d participants in group %s, the mean was %f", N, grp, M)
[1] "For 20 participants in group A, the mean was 14.200000"
```

Format strings erlauben eine weitergehende Formatierung der Ausgabe, indem zwischen dem `%` und dem folgenden Buchstaben Angaben gemacht werden, die sich z. B. auf die Anzahl der auszugebenden Dezimalstellen beziehen können. Für detailliertere Informationen vgl. `?sprintf`.

```
> sprintf("%.3f", 1.23456)      # begrenze Ausgabe auf 3 Dezimalstellen
[1] "1.234"
```

Zeichenketten verbinden

Um mehrere Zeichenketten kombiniert als eine einzige Zeichenkette lediglich auf der R Konsole auszugeben (und nicht in einem Vektor zu speichern), kann die Funktion `cat()` (*concatenate*) verwendet werden. Sie erlaubt auch eine gewisse Formatierung – etwa in Form von Zeilenumbrüchen durch die Escape-Sequenz `\n` oder `\t` für Tabulatoren.

```
> cat("<Zeichenkette 1>", "<Zeichenkette 2>", ..., sep=" ")
```

`cat()` kombiniert die übergebenen Zeichenketten durch Verkettung zunächst zu einer einzelnen, wobei zwischen den Zeichenketten das unter `sep` genannte Trennzeichen eingefügt wird. Numerische Variablen werden hierbei automatisch in Zeichenketten konvertiert. Die Ausgabe von `cat()` unterscheidet sich in zwei Punkten von der üblichen Ausgabe einer Zeichenkette: Zum einen wird sie nicht in Anführungszeichen gesetzt. Zum anderen wird am Anfang jeder Zeile auf die Ausgabe der Position des zu Zeilenbeginn stehenden Wertes, etwa [1], verzichtet.

```
> cVar <- "A string"
> cat(cVar, "with\n", 4, "\nwords\n", sep="+")
A string+with
+4+
words
```

⁴⁷Alternativ kann `fmt` in einfache Anführungszeichen '`(format string)`' gesetzt werden, innerhalb derer sich dann auch doppelte Anführungszeichen ohne voranstehenden backslash \ befinden können (vgl. Abschn. 1.3.5, Fußnote 25).

In der Voreinstellung setzen die meisten Ausgabefunktionen von R Zeichenketten in Anführungszeichen. In `print()` lässt sich dies mit dem Argument `quote=FALSE` verhindern, allgemein hat `noquote("Zeichenkette")` denselben Effekt.

```
> print(cVar, quote=FALSE)
[1] A string
```

```
> noquote(cVar)
[1] A string
```

2.12.3 Zeichenketten manipulieren

`tolower("Z")` und `toupper("Z")` konvertieren die Buchstaben in den übergebenen (Vektoren von) Zeichenketten Z in Klein- bzw. Großbuchstaben. Die Funktion `abbreviate("Z", minlength=Anzahl)` verkürzt Z auf `minlength` viele Buchstaben, was etwa beim Erstellen von übersichtlichen Variablen-Bezeichnungen für Tabellen sinnvoll sein kann.

```
> tolower(c("A", "BC", "DEF"))
[1] "a" "bc" "def"
```

```
> toupper(c("ghi", "jk", "i"))
[1] "GHI" "JK" "I"
```

```
> abbreviate("AfairlyLongString", minlength=6)
AfairlyLongString
"AfrllS"
```

Mit `strsplit()` (*string split*) ist es möglich, eine einzelne Zeichenkette in mehrere Teile zu zerlegen.

```
> strsplit(x="Zeichenkette", split="Zeichenkette", fixed=FALSE)
```

Die Elemente des für x übergebenen Vektors werden dafür nach Vorkommen der unter `split` genannten Zeichenkette durchsucht, die als Trennzeichen interpretiert wird. Die Zeichenfolgen links und rechts von `split` machen die Komponenten der Ausgabe aus, die aus einer Liste von Vektoren von Zeichenketten besteht – eine Komponente für jedes Element des Vektors von Zeichenketten x. In der Voreinstellung `split=NULL` werden die Elemente von x in einzelne Zeichen zerlegt.⁴⁸ `strsplit()` ist damit die Umkehrung von `paste()`. Das Argument `fixed` bestimmt, ob `split` i.S. eines regulären Ausdrucks interpretiert werden soll (Voreinstellung `FALSE`, vgl. Abschn. 2.12.4) oder als exakt die übergebene Zeichenfolge selbst (`TRUE`).

```
> strsplit(c("abc_def_ghi", "jkl_mno"), split="_")
[[1]]
[1] "abc" "def" "ghi"
```

⁴⁸Die Ausgabe ist ggf. mit `unlist()` (vgl. Abschn. 3.1.3) in einen Vektor, oder mit `do.call("cbind", (Liste))` bzw. `do.call("rbind", (Liste))` in eine Matrix umzuwandeln, wenn die Listenkomponenten dieselbe Länge besitzen (vgl. Abschn. 3.4.1).

```
[[2]]
[1] "jkl" "mno"

> strsplit("Xylophon", split=NULL)
[[1]]
[1] "X" "y" "l" "o" "p" "h" "o" "n"
```

Mit `StrRev("<Zeichenkette>")` (*string reverse*) aus dem Paket `DescTools` wird die Reihenfolge der Zeichen innerhalb einer Zeichenkette umgekehrt.

```
> library(DescTools) # für StrRev()
> StrRev(c("Lorem", "ipsum", "dolor", "sit"))
[1] "meroL" "muspi" "rolod" "tis"
```

2.12.4 Zeichenfolgen finden

Die Suche nach bestimmten Zeichenfolgen innerhalb von Zeichenketten ist mit `match()`, `pmatch()` und `grep()` möglich. Soll geprüft werden, ob die in einem Vektor `x` enthaltenen Elemente jeweils eine exakte Übereinstimmung in den Elementen eines Vektors `table` besitzen, ist `match()` anzuwenden. Beide Objekte müssen nicht unbedingt Zeichenketten sein, werden aber intern zu solchen konvertiert.⁴⁹

```
> match(x=<gesuchte Werte>, table=<Objekt>)
```

Die Ausgabe gibt für jedes Element von `x` die erste Position im Objekt `table` an, an der es dort ebenfalls vorhanden ist. Enthält `table` kein mit `x` übereinstimmendes Element, ist die Ausgabe an dieser Stelle NA.

Die fast identische Funktion `pmatch()` unterscheidet sich darin, dass die Elemente von `table` nicht nur auf exakte Übereinstimmung getestet werden: Findet sich für ein Element von `x` ein identisches Element in `table`, ist der Index das Ergebnis, an dem dieses Element zum ersten Mal vorkommt. Andernfalls wird in `table` nach teilweisen Übereinstimmungen in dem Sinne gesucht, dass auch eine Zeichenkette zu einem Treffer führt, wenn sie mit jener aus `x` beginnt, sofern es nur eine einzige solche Zeichenkette in `table` gibt.

```
> match(c("abc", "de", "f", "h"), c("abcde", "abc", "de", "fg", "ih"))
[1] 2 3 NA NA
```

```
> pmatch(c("abc", "de", "f", "h"), c("abcde", "abc", "de", "fg", "ih"))
[1] 2 3 4 NA
```

`grep()` ähnelt dem gleichlautenden POSIX-Befehl Unix-artiger Betriebssysteme und bietet stark erweiterte Suchmöglichkeiten.

```
> grep(pattern=<Suchmuster>, x=<Zeichenkette>)
```

⁴⁹`adist()` berechnet die Levenshtein-Distanz zwischen zwei Zeichenketten als Maß für die Anzahl notwendiger elementarer Editurvorgänge, um eine Zeichenkette in die andere zu ändern. Über die Levenshtein-Distanz können auch ungefähr passende Zeichenketten gefunden werden.

Unter **pattern** ist ein Muster anzugeben, das die zu suchende Zeichenfolge definiert. Obwohl hier auch einfach eine bestimmte Zeichenfolge übergeben werden kann, liegt die Besonderheit darin, dass **pattern** reguläre Ausdrücke akzeptiert. Ein regulärer Ausdruck definiert eine Menge möglicher Zeichenfolgen, die dasselbe Muster besitzen, etwa „ein A gefolgt von einem B oder C und einem Leerzeichen“: "A[BC] [[:blank:]]" (vgl. ?**regex**, Friedl, 2006; Goyvaerts & Levithan, 2012 und speziell für die Anwendung in R Spector, 2008). Der zu durchsuchende Vektor von Zeichenketten wird unter **x** genannt.

Die Ausgabe besteht in einem Vektor von Indizes derjenigen Elemente von **x**, die das gesuchte Muster enthalten. Alternativ gibt die ansonsten genauso zu verwendende Funktion **grep1()** einen logischen Indexvektor aus, der für jedes Element von **x** angibt, ob es **pattern** enthält.

```
> grep("A[BC] [[:blank:]]", c("AB ", "AB", "AC ", "A "))
[1] 1 3
```

```
> grep1("A[BC] [[:blank:]]", c("AB ", "AB", "AC ", "A "))
[1] TRUE FALSE TRUE FALSE
```

2.12.5 Zeichenfolgen extrahieren

Aus Zeichenketten lassen sich mit **substring()** konsekutive Teilstücke von Zeichen extrahieren.

```
> substring(text="Zeichenkette", first=<Beginn>, last=<Ende>)
```

Aus den Elementen des für **text** angegebenen Vektors von Zeichenketten wird jeweils jene Zeichenfolge extrahiert, die beim Buchstaben an der Stelle **first** beginnt und mit dem Buchstaben an der Stelle **last** endet. Umfasst eine Zeichenkette weniger als **first** oder **last** Buchstaben, werden nur so viele ausgegeben, wie tatsächlich vorhanden sind – ggf. eine leere Zeichenkette.

```
> substring(c("ABCDEF", "GHIJK", "LMNO", "PQR"), first=4, last=5)
[1] "DE" "JK" "O" ""
```

Um mit Hilfe von regulären Ausdrücken definierte Zeichenfolgen aus Zeichenketten extrahieren zu können, ist neben der Information, *ob* eine Zeichenkette die gesuchte Zeichenfolge enthält, auch die Information notwendig, an welcher Stelle sie ggf. auftaucht. Dies lässt sich mit **regexp()** ermitteln.

```
> regexp(pattern="Suchmuster", text="Zeichenkette")
```

Die Argumente **pattern** und **text** haben jeweils dieselbe Bedeutung wie **pattern** und **x** von **grep()**. Das Ergebnis ist ein numerischer Vektor mit so vielen Elementen wie jene von **text**. Enthält ein Element von **text** das Suchmuster nicht, ist das Ergebnis an dieser Stelle **-1**. Andernfalls ist das Ergebnis die erste Stelle des zugehörigen Elements von **text**, an der das gefundene Suchmuster dort beginnt. Der ausgegebene numerische Vektor besitzt weiterhin das Attribut **match.length**, das seinerseits ein numerischer Vektor ist und codiert, wie viele Zeichen die Zeichenfolge umfasst, auf die das Suchmuster zutrifft. Auch hier steht die **-1** für den Fall, dass sich das Suchmuster nicht in der Zeichenkette findet. Das Ergebnis eignet sich

besonders, um mit der Funktion `substring()` weiterverarbeitet zu werden, da sich aus ihm die Informationen für deren Argumente `first` und `last` leicht bestimmen lassen.

```
> pat    <- "[[:upper:]]+"          # suche nach Großbuchstaben
> txt    <- c("abcDEFG", "ABCdefg", "abcdefg")
> (start <- regexpr(pat, txt))      # Start und Länge der Fundstellen
[1] 4 1 -1

attr(,"match.length")
[1] 4 3 -1

> len <- attr(start, "match.length") # nur Länge der Fundstellen
> end <- start + len - 1           # letzte Zeichen der Fundstellen
> substring(txt, start, end)       # extrahiere Fundstellen
[1] "DEFG" "ABC" "
```

Im Unterschied zu `regexpr()` berücksichtigt die ansonsten gleich zu verwendende Funktion `gregexpr()` nicht nur das erste Auftreten von `pattern` in `text`, sondern auch ggf. spätere. Die Ausgabe ist eine Liste mit so vielen Komponenten, wie `text` Elemente besitzt.

Eine Alternative zu `substring()` bieten die in Abschn. 2.12.6 vorgestellten Funktionen `sub()` bzw. `gsub()`.

Da die Syntax regulärer Ausdrücke recht komplex ist, sorgt u. U. schon die Suche nach einfachen Mustern Schwierigkeiten. Eine Vereinfachung bietet die Funktion `glob2rx()`, mit der Muster von Zeichenfolgen mit Hilfe gebräuchlicherer Platzhalter (*wildcards* bzw. *Globbing-Muster*) beschrieben und in einen regulären Ausdruck umgewandelt werden können. So steht z. B. der Platzhalter `?` für ein beliebiges einzelnes Zeichen, `*` für eine beliebige Zeichenkette.

```
> glob2rx(pattern="⟨Muster mit Platzhaltern⟩")
```

Das Argument `pattern` akzeptiert einen Vektor, dessen Elemente Zeichenfolgen aus Buchstaben und Platzhaltern sind. Die Ausgabe besteht aus einem Vektor mit regulären Ausdrücken, wie sie z. B. in `grep()` angewendet werden können.

```
> glob2rx("asdf*.txt")    # Namen, die mit asdf beginnen und .txt enden
[1] "^asdf.*\\.txt$"
```

2.12.6 Zeichenfolgen ersetzen

Wenn in Zeichenketten nach bestimmten Zeichenfolgen gesucht wird, dann häufig, um sie durch andere zu ersetzen. Dies ist etwa möglich, indem dem Ergebnis von `substring()` ein passender Vektor von Zeichenketten zugewiesen wird – dessen Elemente ersetzen dann die durch `first` und `last` begrenzten Zeichenfolgen in den Elementen von `text`. Dabei ist es notwendig, dass für `text` ein bereits bestehendes Objekt übergeben wird, das dann der Änderung unterliegt.

```
> charVec <- c("ABCDEF", "GHIJK", "LMNO", "PQR")
> substring(charVec, 4, 5) <- c(..", "xx", "++", "**")
> charVec
[1] "ABC..F" "GHIxx" "LMN+" "PQR"
```

Auch `sub()` (*substitute*) und `gsub()` dienen dem Zweck, durch ein Muster definierte Zeichenfolgen innerhalb von Zeichenketten auszutauschen.

```
> sub(pattern="<Suchmuster>", replacement="<Ersatz>",
+     x="<Zeichenkette>")
```

Für `pattern` kann ein regulärer Ausdruck übergeben werden, dessen Vorkommen in den Elementen von `x` durch die unter `replacement` genannte Zeichenfolge ersetzt werden. Wenn `pattern` in einem Element von `x` mehrfach vorkommt, wird es nur beim ersten Auftreten ersetzt.

```
> sub("em", "XX", "Lorem ipsum dolor sit Lorem ipsum")
[1] "LorXX ipsum dolor sit Lorem ipsum"
```

Im Unterschied zu `sub()` ersetzt die ansonsten gleich zu verwendende `gsub()` Funktion `pattern` nicht nur beim ersten Auftreten in `x` durch `replacement`, sondern überall.

```
> gsub("em", "XX", "Lorem ipsum dolor sit Lorem ipsum")
[1] "LorXX ipsum dolor sit LorXX ipsum"
```

In `pattern` können runde Klammern (*<Muster>*) zur Definition von Gruppen innerhalb des regulären Ausdrucks verwendet werden. Die Gruppen sind intern numeriert. Zeichenketten, die das Muster einer Gruppe aufweisen, sind dann innerhalb von `replacement` über die Angabe der Gruppen-Nummer in der Form "`\(\Nummer)`" abrufbar (*back referencing*). Auf diese Weise lassen sich Zeichenfolgen etwas einfacher als mit `substring()` (vgl. Abschn. 2.12.5) extrahieren.

Im Beispiel definiert der reguläre Ausdruck eine Zeichenkette, die mit einem oder mehr Buchstaben beginnt, dann in - eingeschlossen eine Gruppe von einer oder mehr Ziffern aufweist und mit einem oder mehr Buchstaben endet. Die Zifferngruppe wird durch back referencing extrahiert.

```
> gsub("^[:alpha:]+-[:digit:]+-[:alpha:]+$", "\1", "abc-412-def")
[1] "412"
```

2.12.7 Zeichenketten als Befehl ausführen

Durch die Kombination von `parse()` und `eval()` lassen sich Zeichenketten als Befehle interpretieren und wie direkt eingegebene Befehle ausführen. Dieses Zusammenspiel ermöglicht es, in Abhängigkeit von vorherigen Auswertungen einen nachfolgend benötigten Befehl zunächst als Zeichenkette zu erstellen und dann auszuführen.

```
> parse(file="<Pfad und Dateiname>", text="<Zeichenkette>")
```

Hierfür ist zunächst mit `parse()` eine für das Argument `text` zu übergebende Zeichenkette in ein weiter interpretierbares Objekt umzuwandeln.⁵⁰ Ist `text` ein Vektor von Zeichenketten, wird jedes Element als ein Befehl verstanden. Alternativ kann mit `file` eine Datei oder sonstige Quelle genannt werden, die eine solche Zeichenkette enthält (vgl. Abschn. 4.2.2).

⁵⁰Solcherart erstellte Objekte können mit `deparse()` wieder in Zeichenketten umgewandelt werden.

```
> obj1 <- parse(text="3 + 4")
> obj2 <- parse(text=c("vec <- c(1, 2, 3)", "vec^2"))
```

Das Ausführen eines mit `parse()` erstellten Objekts geschieht mit `eval(<expression>)`.

```
> eval(obj1)
[1] 7
```

```
> eval(obj2)
[1] 1 4 9
```

2.13 Datum und Uhrzeit

Insbesondere bei der Analyse von Zeitreihen⁵¹ ist es sinnvoll, Zeit- und Datumsangaben in einer Form zu speichern, die es erlaubt, solche Werte in natürlicher Art für Berechnungen zu nutzen – etwa um über die Differenz zweier Uhrzeiten die zwischen ihnen verstrichene Zeit ebenso zu ermitteln wie die zwischen zwei Datumsangaben liegende Anzahl von Tagen. R bietet solche Möglichkeiten mit Hilfe besonderer Klassen.⁵²

2.13.1 Datumsangaben erstellen und formatieren

Objekte der Klasse `Date` codieren ein Datum mittels der seit einem Stichtag (meist der 1. Januar 1970) verstrichenen Anzahl von Tagen und können Tag, Monat und Jahr eines Zeitpunkts ausgeben. Das aktuelle Datum unter Beachtung der Zeitzone nennt `Sys.Date()` in Form eines `Date` Objekts. Um selbst ein Datum zu erstellen, ist `as.Date()` zu verwenden.

```
> as.Date(x="", format="")
```

Die Datumsangabe für `x` ist eine Zeichenkette, die ein Datum in einem Format nennt, das unter `format` als `format string` zu spezifizieren ist. In einer solchen Zeichenkette stehen `%<Buchstabe>` Kombinationen als Platzhalter für den einzusetzenden Teil einer Datumsangabe, sonstige Zeichen i. d. R. für sich selbst. Voreinstellung ist `"%Y-%m-%d"`, wobei `%Y` für die vierstellige Jahreszahl, `%m` für die zweistellige Zahl des Monats und `%d` für die zweistellige Zahl der Tage steht.⁵³ In diesem Format erfolgt auch die Ausgabe, die sich jedoch mit `format(<Date-Objekt>, format="")` kontrollieren lässt.

⁵¹Für die Auswertung von Zeitreihen vgl. Shumway und Stoffer (2011) sowie den Abschnitt *Time Series Analysis* der CRAN Task Views (Hyndman, 2014).

⁵²Für eine einführende Behandlung der vielen für Zeitangaben existierenden Subtilitäten vgl. Grothendieck und Petzoldt (2004) sowie `?DateTimeClasses`. Der Umgang mit Zeit- und Datumsangaben wird durch Funktionen des Pakets `lubridate` (Grolmusz & Wickham, 2011) erleichtert. Das Paket `timeDate` (Würtz, Chalabi & Maechler, 2013) enthält viele weiterführende Funktionen zur Verarbeitung solcher Daten.

⁵³Vergleiche Abschn. 2.12.2 sowie `?strptime` für weitere mögliche Elemente des `format strings`. Diese Hilfe-Seite erläutert auch, wie mit Namen für Wochentage und Monate in unterschiedlichen Sprachen umzugehen ist.

```
> Sys.Date()
[1] "2009-02-09"

> (myDate <- as.Date("01.11.1974", format="%d.%m.%Y"))
[1] "1974-11-01"

> format(myDate, format="%d.%m.%Y")
[1] "01.11.1974"
```

Ihre numerische Repräsentation lässt sich direkt zum Erstellen von **Date** Objekten nutzen.

```
> as.Date(x=<Vektor>, origin=<Stichtag>)
```

Für **x** ist ein numerischer Vektor mit der seit dem Stichtag **origin** verstrichenen Anzahl von Tagen zu nennen. Negative Zahlen stehen dabei für die Anzahl der Tage vor dem Stichtag. Der Stichtag selbst muss in Form eines **Date** Objekts angegeben werden.

```
# Datum, das 374 Tage vor dem 16.12.1910 liegt
> (negDate <- as.Date(-374, "1910-12-16"))
[1] "1909-12-07"
```

```
> as.numeric(negDate) # Anzahl Tage vor Standard-Stichtag
[1] -21940
```

2.13.2 Uhrzeit

Objekte der Klasse **POSIXct** (*calendar time*) repräsentieren neben dem Datum gleichzeitig die Uhrzeit eines Zeitpunkts als Anzahl der Sekunden, die seit einem Stichtag (meist der 1. Januar 1970) verstrichen ist, besitzen also eine Genauigkeit von einer Sekunde. Negative Zahlen stehen dabei für die Anzahl der Sekunden vor dem Stichtag. **POSIXct** Objekte berücksichtigen die Zeitzone sowie die Unterscheidung von Sommer- und Winterzeit.

Sys.time() gibt das aktuelle Datum nebst Uhrzeit in Form eines **POSIXct** Objekts aus, allerdings für eine Standard-Zeitzone. Das Ergebnis muss deshalb mit den u. g. Funktionen in die aktuelle Zeitzone konvertiert werden. Alternativ gibt **date()** Datum und Uhrzeit mit englischen Abkürzungen für Wochentag und Monat als Zeichenkette aus, wobei die aktuelle Zeitzone berücksichtigt wird.

```
> Sys.time()
[1] "2009-02-07 09:23:02 CEST"

> date()
[1] "Sat Feb 7 09:23:02 2009"
```

Objekte der Klasse **POSIXlt** (*local time*) speichern dieselbe Information, allerdings nicht in Form der seit einem Stichtag verstrichenen Sekunden, sondern als Liste mit benannten Komponenten: Dies sind numerische Vektoren u. a. für die Sekunden (**sec**), Minuten (**min**) und Stunden (**hour**) der Uhrzeit sowie für Tag (**mday**), Monat (**mon**) und Jahr (**year**) des Datums. Zeichenketten

lassen sich analog zu `as.Date()` mit `as.POSIXct()` bzw. mit `as.POSIXlt()` in entsprechende Objekte konvertieren, `strptime()` erzeugt ebenfalls ein `POSIXlt` Objekt.

```
> as.POSIXct(x="Datum und Uhrzeit", format="(format string)")
> as.POSIXlt(x="Datum und Uhrzeit", format="(format string)")
>   strptime(x="Datum und Uhrzeit", format="(format string))
```

Voreinstellung für den format string bei `as.POSIXlt()` und bei `as.POSIXct()` ist "`%Y-%m-%d %H:%M:%S`", wobei `%H` für die zweistellige Zahl der Stunden im 24 h-Format, `%M` für die zweistellige Zahl der Minuten und `%S` für die zweistellige Zahl der Sekunden des Datums stehen (vgl. Fußnote 53).

```
> (myTime <- as.POSIXct("2009-02-07 09:23:02"))
[1] "2009-02-07 09:23:02 CET"

> charDates <- c("05.08.1972, 03:37", "31.03.1981, 12:44")
> lDates   <- strptime(charDates, format="%d.%m.%Y, %H:%M")
[1] "1972-08-05 03:37:00" "1981-03-31 12:44:00"

> lDates$mday                         # Tag isoliert
[1] 5 31

> lDates$hour                         # Stunde isoliert
[1] 3 12
```

`POSIXct` Objekte können besonders einfach mit der Funktion `ISOdate()` erstellt werden, die intern auf `strptime()` basiert, aber keinen format string benötigt.

```
> ISOdate(year=<Jahr>, month=<Monat>, day=<Tag>,
+           hour=<Stunde>, min=<Minute>, sec=<Sekunde>, tz="Zeitzone")
```

Für die sich auf Datum und Uhrzeit beziehenden Argumente können Zahlen im 24 h-Format angegeben werden, wobei 12:00:00 h Voreinstellung für die Uhrzeit ist. Mit `tz` lässt sich die Zeitzone im Form der standardisierten Akronyme festlegen, Voreinstellung ist hier "GMT".

```
# Zeitzone: Central European Time
> ISOdate(2010, 6, 30, 17, 32, 10, tz="CET")
[1] "2010-06-30 17:32:10 CEST"
```

Auch Objekte der Klassen `POSIXct` und `POSIXlt` können mit `format()` in der gewünschten Formatierung ausgegeben werden.

```
> format(myTime, "%H:%M:%S")          # nur Stunden, Minuten, Sekunden
[1] "09:23:02"

> format(lDates, "%d.%m.%Y")         # nur Tag, Monat, Jahr
[1] "05.08.1972" "31.03.1981"
```

Aus Datumsangaben der Klasse `Date`, `POSIXct` und `POSIXlt` lassen sich bestimmte weitere Informationen in Form von Zeichenketten extrahieren, etwa der Wochentag mit `weekdays(<Datum>)`, der Monat mit `months(<Datum>)` oder das Quartal mit `quarters(<Datum>)`.

```
> weekdays(lDates)
[1] "Samstag" "Dienstag"

> months(lDates)
[1] "August" "März"
```

2.13.3 Mit Datum und Uhrzeit rechnen

Objekte der Klasse `Date`, `POSIXct` und `POSIXlt` verhalten sich in vielen arithmetischen Kontexten in natürlicher Weise, da die sinnvoll für Daten interpretierbaren Rechenfunktionen besondere Methoden für sie besitzen (vgl. Abschn. 15.2.6): So werden zu `Date` Objekten addierte Zahlen als Anzahl von Tagen interpretiert; das Ergebnis ist ein Datum, das entsprechend viele Tage vom `Date` Objekt abweicht. Die Differenz zweier `Date` Objekte besitzt die Klasse `difftime` und wird als Anzahl der Tage im Zeitintervall vom zweiten zum ersten Datum ausgegeben. Hierbei ergeben sich negative Zahlen, wenn das erste Datum zeitlich vor dem zweiten liegt.⁵⁴ Ebenso wie Zahlen lassen sich auch `difftime` Objekte zu `Date` Objekten addieren.

```
> myDate + 365
[1] "1975-11-01"

> (diffDate <- as.Date("1976-06-19") - myDate) # Zeitintervall
Time difference of 596 days

> as.numeric(diffDate)                                # Intervall als Zahl
[1] 596

> myDate + diffDate
[1] "1976-06-19"
```

Zu Objekten der Klasse `POSIXlt` oder `POSIXct` addierte Zahlen werden als Sekunden interpretiert. Aus der Differenz zweier solcher Objekte entsteht ebenfalls ein Objekt der Klasse `difftime`. Die Addition von `difftime` und `POSIXlt` oder `POSIXct` Objekten ist ebenfalls definiert.

```
> lDates + c(60, 120)                               # 1 und 2 Minuten später
[1] "1972-08-05 03:38:00 CET" "1981-03-31 12:46:00 CEST"

> (diff21 <- lDates[2] - lDates[1])
Time difference of 3160.338 days

> lDates[1] + diff21
[1] "1981-03-31 12:44:00 CEST"
```

In `seq()` (vgl. Abschn. 2.4.1) ändert sich die Bedeutung des Arguments `by` hin zu Zeitangaben, wenn für `from` und `to` Datumsangaben übergeben werden. Für die Schrittweite werden dann

⁵⁴Die Zeiteinheit der im `difftime` Objekt gespeicherten Werte (etwa Tage oder Minuten), hängt davon ab, aus welchen Datumsangaben das Objekt entstanden ist. Alternativ bestimmt das Argument `units` von `difftime()`, um welche Einheit es sich handeln soll.

etwa die Werte "`<Anzahl> years`" oder "`<Anzahl> days`" akzeptiert (vgl. `?seq.POSIXt`). Dies gilt analog auch für das Argument `breaks` der `cut()` Funktion (vgl. Abschn. 2.6.7), die kontinuierliche Daten in Kategorien einteilt, die etwa durch Stunden (`breaks="hour"`) oder Kalenderwochen (`breaks="week"`) definiert sind (vgl. `?cut.POSIXt`). Für weitere geeignete arithmetische Funktionen vgl. `methods(class="POSIXt")` und `methods(class="Date")`.

```
# jährliche Schritte vom 01.05.2010 bis zum 01.05.2013
> seq(ISOdate(2010, 5, 1), ISOdate(2015, 5, 1), by="years")
[1] "2010-05-01 12:00:00 GMT" "2011-05-01 12:00:00 GMT"
[3] "2012-05-01 12:00:00 GMT" "2013-05-01 12:00:00 GMT"

# 4 zweiwöchentliche Schritte vom 22.10.1997
> seq(ISOdate(1997, 10, 22), by="2 weeks", length.out=4)
[1] "1997-10-22 12:00:00 GMT" "1997-11-05 12:00:00 GMT"
[3] "1997-11-19 12:00:00 GMT" "1997-12-03 12:00:00 GMT"

# 100 zufällige Daten zwischen 13.06.1995 und 4 Wochen später
> secsPerDay <- 60 * 60 * 24 # Sekunden pro Tag
> randDates <- ISOdate(1995, 6, 13)
+           + sample(0:(28*secsPerDay), 100, replace=TRUE)

# teile Daten in Kalenderwochen ein
> randWeeks <- cut(randDates, breaks="week")
> summary(randWeeks) # Häufigkeiten
1995-06-12 1995-06-19 1995-06-26 1995-07-03 1995-07-10
      15          26          20          37          2
```

Kapitel 3

Datensätze

Vektoren, Matrizen und arrays sind dahingehend eingeschränkt, dass sie gleichzeitig nur Werte desselben Datentyps aufnehmen können. Da in empirischen Erhebungen meist Daten unterschiedlichen Typs – etwa numerische Variablen, Faktoren und Zeichenketten – anfallen, sind sie nicht unmittelbar geeignet, vollständige Datensätze zu speichern. Objekte der Klasse `list` und `data.frame` sind in dieser Hinsicht flexibler: Sie erlauben es, gleichzeitig Variablen unterschiedlichen Datentyps und auch unterschiedlicher Klasse als Komponenten zu besitzen.

Der Datentyp von Listen und Datensätzen selbst ist `list`. Listen eignen sich zur Repräsentation heterogener Sammlungen von Daten und werden deshalb von vielen Funktionen genutzt, um ihr Ergebnis zurückzugeben. Listen sind darüber hinaus die allgemeine Grundform von Datensätzen (Klasse `data.frame`), der gewöhnlich am besten geeigneten Struktur für empirische Daten.

3.1 Listen

Listen werden mit dem Befehl `list(<Komponente1>, <Komponente2>, ...)` erzeugt, wobei für jede Komponente ein (ggf. bereits bestehendes) Objekt zu nennen ist. Alternativ lässt sich eine Liste mit `<Anzahl>` vielen leeren Komponenten über `vector("list", <Anzahl>)` erstellen. Komponenten einer Liste können Objekte jeglicher Klasse und jedes Datentyps, also auch selbst wieder Listen sein. Die erste Komponente könnte also z. B. ein numerischer Vektor, die zweite ein Vektor von Zeichenketten und die dritte eine Matrix aus Wahrheitswerten sein. Die von `length(<Liste>)` ausgegebene Länge einer Liste ist die Anzahl ihrer Komponenten auf oberster Ebene.

```
> myList1 <- list(c(1, 3), c(12, 8, 29, 5))    # Liste erstellen
> length(myList1)                                # Anzahl Komponenten
[1] 2

> vector("list", 2)                               # Liste: 2 leere Komponenten
[[1]]
NULL

[[2]]
NULL
```

3.1.1 Komponenten auswählen und verändern

Um auf eine Listen-Komponente zuzugreifen, kann der `[[<Index>]]` Operator benutzt werden, der als Argument die Position der zu extrahierenden Komponente in der Liste benötigt. Dabei kann der Index auch in einem Objekt gespeichert sein. `[[<Index>]]` gibt immer nur eine Komponente zurück, selbst wenn mehrere Indizes in Form eines Indexvektors übergeben werden.¹ Die zweite Komponente einer Liste könnte also so ausgelesen werden:

```
> myList1[[2]]
[1] 12 8 29 5
```

```
> idx <- 1
> myList1[[idx]]
[1] 1 3
```

Einzelne Elemente einer aus mehreren Werten bestehenden Komponente können auch direkt abgefragt werden, etwa das dritte Element des obigen Vektors. Dazu wird der für Vektoren genutzte `[<Index>]` Operator an den Listenindex `[[<Index>]]` angehängt, weil die Auswertung des Befehls `myList1[[2]]` zuerst erfolgt und den im zweiten Schritt zu indizierenden Vektor zurückliefert:

```
> myList1[[2]][3]
[1] 29
```

Beispiel sei eine Liste aus drei Komponenten. Die erste soll ein Vektor sein, die zweite eine aus je zwei Zeilen und Spalten bestehende Matrix, die dritte ein Vektor aus Zeichenketten.

```
> (myList2 <- list(1:4, matrix(1:4, 2, 2), c("Lorem", "ipsum")))
[[1]]
[1] 1 2 3 4

[[2]]
[,1] [,2]
[1,]    1    3
[2,]    2    4

[[3]]
[1] "Lorem" "ipsum"
```

Das Element in der ersten Zeile und zweiten Spalte der Matrix, die ihrerseits die zweite Komponente der Liste darstellt, wäre dann so aufzurufen:

```
> myList2[[2]][1, 2]
[1] 3
```

Auch bei Listen kann der `[<Index>]` Operator verwendet werden, der immer ein Objekt desselben Datentyps zurückgibt wie den des indizierten Objekts. Im Unterschied zu `[[<Index>]]` liefert er deshalb nicht die Komponente selbst zurück, sondern eine Liste, die wiederum als einzige

¹Für Hilfe zu diesem Thema vgl. `?Extract`.

Komponente das gewünschte Objekt besitzt – also gewissermaßen eine Teilliste ist. Während `myList2[[2]]` also eine numerische Matrix ausgibt, ist das Ergebnis von `myList2[2]` eine Liste, deren einzige Komponente diese Matrix ist.

```
> myList2[[2]]          # Komponente: numerische Matrix -> Datentyp numeric
   [,1]  [,2]
[1,]    1    3
[2,]    2    4

> mode(myList2[[2]])
[1] "numeric"

> myList2[2]           # Teilliste -> Datentyp list
[[1]]
   [,1]  [,2]
[1,]    1    3
[2,]    2    4

> mode(myList2[2])
[1] "list"
```

Wie auch bei Vektoren können die Komponenten einer Liste benannt sein und mittels `\(Liste)[["\<Variablenname\>"]]` über ihren Namen ausgewählt werden.

```
> (myList3 <- list(numvec=1:5, word="dolor"))
$numvec
[1] 1 2 3 4 5

$word
[1] "dolor"

> myList3[["word"]]
[1] "dolor"
```

Wenn man auf eine benannte Komponente zugreifen will, kann dies mittels `[[[]]]` Operator und ihrem numerischen Index oder ihrem Namen geschehen, zusätzlich aber auch über den Operator `\(Liste)\$<Variablenname>`. Dieser bietet den Vorteil, dass er ohne Klammern und numerische Indizes auskommt und damit recht übersichtlich ist. Nur wenn der Name Leerzeichen enthält, wie es bisweilen bei von R zurückgegebenen Objekten der Fall ist, muss er zudem in Anführungszeichen stehen. `$` benötigt immer den Variablennamen selbst, anders als mit `[[[]]]` kann kein Objekt verwendet werden, das den Namen speichert. Welche Namen die Komponenten tragen, erfährt man mit `names(\(Liste))`.

```
> myList3$numvec
[1] 1 2 3 4 5

> mat      <- cbind(1:10, sample(-10:10, 10, replace=FALSE))
> retList <- cov.wt(mat, method="ML")      # unkorrigierte Kovarianzmatrix
> names(retList)                         # Komponenten der Liste
```

```
[1] "cov" "center" "n.obs"

> retList$cov                                # Kovarianzmatrix selbst
[1,] [,1] [,2]
[1,] 8.25  6.20
[2,] 6.20  28.44

> retList$center                             # Spaltenmittel
[1] 5.5 3.4

> retList$n.obs                            # Anzahl Beobachtungen
[1] 10
```

3.1.2 Komponenten hinzufügen und entfernen

Auf dieselbe Weise, wie sich die Komponenten einer Liste anzeigen lassen, können auch weitere Komponenten zu einer bestehenden Liste hinzugefügt werden, also mit

- `<Liste>[[<Index>]]`
- `<Liste>[["<neue Komponente>"]]`
- `<Liste>$<neue Komponente>`

```
> myList1[[3]]           <- LETTERS[1:5]    # 1. neue Komponente
> myList1[["neuKomp2"]] <- letters[1:5]     # 2. neue Komponente
> myList1$neuKomp3      <- 100:105       # 3. neue Komponente
> myList1
[[1]]
[1] 1 3

[[2]]
[1] 12 8 29 5

[[3]]
[1] "A" "B" "C" "D" "E"
```

```
$neuKomp2
[1] "a" "b" "c" "d" "e"
```

```
$neuKomp3
[1] 100 101 102 103 104 105
```

Um die Komponenten mehrerer Listen zu einer Liste zu verbinden, eignet sich wie bei Vektoren `c(<Liste1>, <Liste2>, ...)`.

```
> myListJoin <- c(myList1, myList2)      # verbinde Listen
```

Komponenten einer Liste werden gelöscht, indem ihnen die leere Menge `NULL` zugewiesen wird.

```
myList1$neuKomp3 <- NULL # lösche Komponente
```

3.1.3 Listen mit mehreren Ebenen

Da Komponenten einer Liste Objekte verschiedener Klassen und auch selbst Listen sein können, ergibt sich die Möglichkeit, Listen zur Repräsentation hierarchisch organisierter Daten unterschiedlicher Art zu verwenden. Derartige Objekte können auch als Baum mit mehreren Verästelungsebenen betrachtet werden. Um aus einem solchen Objekt einen bestimmten Wert zu erhalten, kann man sich zunächst mit `str(<Liste>)` einen Überblick über die Organisation der Liste verschaffen und sich ggf. sukzessive von Ebene zu Ebene zum gewünschten Element vorarbeiten.

Im Beispiel soll aus einer Liste mit letztlich drei Ebenen ein Wert aus einer Matrix extrahiert werden, die sich in der dritten Verästelungsebene befindet.

```
# 4 Komponenten in der zweiten Ebene
> myListAA <- list(AAA=c(1, 2), AAB=c("AAB1", "AAB2", "AAB3"))
> myMatAB <- matrix(1:8, nrow=2)
> myListBA <- list(BAA=matrix(rnorm(10), ncol=2), BAB=c("BAB1", "BAB2"))
> myVecBB <- sample(1:10, 5)

# 2 Komponenten in der ersten Ebene
> myListA <- list(AA=myListAA, AB=myMatAB)
> myListB <- list(BA=myListBA, BB=myVecBB)

# Gesamtliste
> myList4 <- list(A=myListA, B=myListB)
> str(myList4) # Gesamtstruktur
List of 2
$ A:List of 2
..$ AA:List of 2
... .$ AAA: num [1:2] 1 2
... .$ AAB: chr [1:3] "AAB1" "AAB2" "AAB3"
..$ AB: int [1:2, 1:4] 1 2 3 4 5 6 7 8

$ B:List of 2
..$ BA:List of 2
... .$ BAA: num [1:5, 1:2] -0.618 -1.059 -1.150 0.919 -1.146 ...
... .$ BAB: chr [1:2] "BAB1" "BAB2"
..$ BB: int [1:5] 3 5 7 2 9

# Struktur der 2. Komponente (B) der 1. Ebene
> str(myList4$B)
List of 2
$ BA:List of 2
```

```

..$ BAA: num [1:5, 1:2] -0.618 -1.059 -1.150 0.919 -1.146 ...
..$ BAB: chr [1:2] "BAB1" "BAB2"
$ BB: int [1:5] 3 5 7 2 9

# Struktur der 3. Komponente (BA) der 2. Ebene
> str(myList4$B$BA)
List of 2
$ BAA: num [1:5, 1:2] -0.618 -1.059 -1.150 0.919 -1.146 ...
$ BAB: chr [1:2] "BAB1" "BAB2"

# Element der 1. Komponente (BAA) der 3. Ebene
> myList4$B$BA$BAA[4, 2]
[1] 0.1446770

```

Die hierarchische Struktur einer Liste kann mit dem Befehl `unlist(<Liste>)` aufgelöst werden. Der Effekt besteht darin, dass alle Komponenten (in der Voreinstellung `recursive=TRUE` rekursiv, d.h. einschließlich aller Ebenen) in denselben Datentyp umgewandelt und seriell in einem Vektor zusammengefügt werden. Als Datentyp wird jener gewählt, der alle Einzelwerte ohne Informationsverlust speichern kann (vgl. Abschn. 1.3.5).

```

> myList5 <- list(c(1, 2, 3), c("A", "B"), matrix(5:12, 2))
> unlist(myList5)
[1] "1" "2" "3" "A" "B" "5" "6" "7" "8" "9" "10" "11" "12"

```

3.2 Datensätze

Ein Datensatz ist eine spezielle Liste und besitzt die Klasse `data.frame`. Datensätze erben die Grundeigenschaften einer Liste, besitzen aber bestimmte einschränkende Merkmale – so müssen ihre Komponenten alle dieselbe Länge besitzen. Die in einem Datensatz zusammengefassten Objekte können von unterschiedlicher Klasse sein und Werte unterschiedlichen Datentyps beinhalten. Dies entspricht der empirischen Situation, dass Werte verschiedener Variablen an derselben Menge von Beobachtungsobjekten erhoben wurden. Anders gesagt enthält jede einzelne Variable Werte einer festen Menge von Beobachtungsobjekten, die auch die Werte für die übrigen Variablen geliefert haben. Werte auf unterschiedlichen Variablen lassen sich somit einander hinsichtlich des Beobachtungsobjekts zuordnen, von dem sie stammen. Da Datensätze gewöhnliche Objekte sind, ist es im Gegensatz zu einigen anderen Statistikprogrammen möglich, mit mehreren von ihnen gleichzeitig zu arbeiten.

Die Basisinstallation von R beinhaltet bereits viele vorbereitete Datensätze, an denen sich statistische Auswertungsverfahren erproben lassen, vgl. `data()` für eine Übersicht. Weitere Datensätze werden durch Zusatzpakete bereitgestellt und lassen sich mit `data(<Datensatz>, package="<Paketname>")` laden.² Nähtere Erläuterungen zu einem dokumentierten Datensatz gibt `help(<Datensatz>)` aus.

²Hervorzuheben sind etwa **DAAG** (Maindonald & Braun, 2014) und **HSAUR2** (Everitt & Hothorn, 2014).

Objekte der Klasse `data.frame` sind die bevorzugte Organisationsweise für empirische Datensätze.³ Listenkomponenten spielen dabei die Rolle von Variablen und ähneln damit den Spalten einer Matrix. Werden Datensätze in R ausgegeben, stehen die Variablen als Komponenten in den Spalten, während jede Zeile i. d. R. für ein Beobachtungsobjekt steht. Datensätze werden mit `data.frame(<Objekt1>, <Objekt2>, ...)` aus mehreren einzelnen Objekten erzeugt, die typischerweise Vektoren oder Faktoren sind. Matrizen werden dabei wie separate, durch die Spalten der Matrix gebildete Vektoren behandelt.⁴

Als Beispiel seien 12 Personen betrachtet, die zufällig auf drei Untersuchungsgruppen (Kontrollgruppe CG, Wartelisten-Gruppe WL, Treatment-Gruppe T) verteilt werden. Als Variablen werden demografische Daten, Ratings und der IQ-Wert simuliert. Zudem soll die fortlaufende Nummer jeder Person gespeichert werden.⁵

```
> N      <- 12
> sex    <- sample(c("f", "m"), N, replace=TRUE)
> group  <- sample(rep(c("CG", "WL", "T"), 4), N, replace=FALSE)
> age    <- sample(18:35, N, replace=TRUE)
> IQ     <- round(rnorm(N, mean=100, sd=15))
> rating <- round(runif(N, min=0, max=6))
> (myDf1 <- data.frame(id=1:N, sex, group, age, IQ, rating))
   id sex group age  IQ rating
1  1   f     T  26 112      1
2  2   m    CG  30 122      3
3  3   m    CG  25  95      5
4  4   m     T  34 102      5
5  5   m    WL  22  82      2
6  6   f    CG  24 113      0
7  7   m     T  28  92      3
8  8   m    WL  35  90      2
9  9   m    WL  23  88      3
10 10  m    WL  29  81      5
11 11  m    CG  20  92      1
12 12  f     T  21  98      1
```

In der ersten Spalte der Ausgabe befinden sich die Zeilennamen, die in der Voreinstellung mit den Zeilennummern übereinstimmen. Eine spätere Teilauswahl der Zeilen (vgl. Abschn. 3.3.3) hebt diese Korrespondenz jedoch häufig auf.

Die Anzahl von Beobachtungen (Zeilen) und Variablen (Spalten) kann wie bei Matrizen mit `dim(<Datensatz>)`, `nrow(<Datensatz>)` und `ncol(<Datensatz>)` ausgegeben werden. Die mit `length()` ermittelte Länge eines Datensatzes ist die Anzahl der in ihm gespeicherten Variablen,

³ Außer bei sehr großen Datensätzen, die sich effizienter als Matrix oder als spezielle Datenstruktur verarbeiten lassen, wie sie das Paket `data.table` (Dowle, Short, Lianoglou & Srinivasan, 2014) bereitstellt.

⁴ Gleicher gilt für Listen – hier werden die Komponenten als separate Vektoren gewertet. Soll dieses Verhalten verhindert werden, um eine Liste als eine einzelne Variable des Datensatzes zu erhalten, muss sie in `I()` eingeschlossen werden: `data.frame(I(<Liste>), <Objekt2>, ...)`.

⁵ Für die automatisierte Simulation von Datensätzen nach vorgegebenen Kriterien, etwa hinsichtlich der Gruppen-Effekte, vgl. die `sim.<Typ>()` Funktionen des Pakets `psych` (Revelle, 2014).

also Spalten. Eine Übersicht über die Werte aller Variablen eines Datensatzes erhält man durch `summary(⟨Datensatz⟩)`.

```
> dim(myDf1)
[1] 12 6

> nrow(myDf1)
[1] 12

> ncol(myDf1)
[1] 6

> summary(myDf1)
      id   sex group       age        IQ      rating
Min. : 1.00 m:9  CG:4  Min. :20.00  Min. :81.00  Min. :0.000
1st Qu.: 3.75 w:3  T :4  1st Qu.:22.75  1st Qu.:89.50  1st Qu.:1.000
Median : 6.50    WL:4 Median :25.50  Median :93.50  Median :2.500
Mean   : 6.50          Mean :26.42  Mean  :97.25  Mean  :2.583
3rd Qu.: 9.25          3rd Qu.:29.25 3rd Qu.:104.50 3rd Qu.:3.500
Max.  :12.00          Max. :35.00  Max. :122.00 Max. :5.000
```

Will man sich einen Überblick über die in einem Datensatz gespeicherten Werte verschaffen, können die Funktionen `head(⟨Datensatz⟩, n=⟨Anzahl⟩)` und `tail(⟨Datensatz⟩, n=⟨Anzahl⟩)` verwendet werden, die seine ersten bzw. letzten n Zeilen anzeigen. Mit `View(⟨Datensatz⟩)` ist es zudem möglich, ein separates Fenster – in RStudio ein Tab – mit dem Inhalt eines Datensatzes zu öffnen. Dessen Werte sind dabei vor Veränderungen geschützt.

3.2.1 Datentypen in Datensätzen

Mit `str(⟨Datensatz⟩)` kann die interne Struktur des Datensatzes erfragt werden, d. h. aus welchen Gruppierungsfaktoren und wie vielen Beobachtungen an welchen Variablen er besteht.

```
> str(myDf1)
'data.frame': 12 obs. of 6 variables:
 $ id   : int 1 2 3 4 5 6 7 8 9 10 ...
 $ sex  : Factor w/ 2 levels "f", "m": 1 2 2 2 2 1 2 2 2 2 ...
 $ group: Factor w/ 3 levels "CG", "T", "WL": 2 1 1 2 3 1 2 ...
 $ age   : int 26 30 25 34 22 24 28 35 23 29 ...
 $ IQ    : num 112 122 95 102 82 113 92 90 88 81 ...
 $ rating: num 1 3 5 5 2 0 3 2 3 5 ...
```

In der Struktur ist hier zu erkennen, dass die in den Datensatz aufgenommenen `character` Vektoren automatisch in Objekte der Klasse `factor` umgewandelt werden. Eine Umwandlung von Zeichenketten in `factor` Objekte ist nicht immer gewünscht, denn Zeichenketten codieren nicht immer eine Gruppenzugehörigkeit, sondern können auch andere Variablen wie Namen darstellen. Wird ein Zeichenketten-Vektor in `I(⟨Vektor⟩)` eingeschlossen, verhindert dies die automatische Umwandlung. Für alle `character` Vektoren gleichzeitig kann dies auch über das `stringsAsFactors=FALSE` Argument von `data.frame()` erreicht werden.

```
> fac    <- c("CG", "T1", "T2")
> DV1   <- c(14, 22, 18)
> DV2   <- c("red", "blue", "blue")
> myDf2 <- data.frame(fac, DV1, DV2, stringsAsFactors=FALSE)
> str(myDf2)
'data.frame': 3 obs. of 3 variables:
$ fac: chr  "CG" "T1" "T2"
$ DV1: num  14 22 18
$ DV2: chr  "red" "blue" "blue"
```

Im Beispiel wird dabei allerdings auch der Vektor `fac` nicht mehr als Faktor interpretiert. Stellt nur einer von mehreren `character` Vektoren eine Gruppierungsvariable dar, so kann das Argument `stringsAsFactors=FALSE` zwar verwendet werden, die Gruppierungsvariable ist dann aber vor oder nach der Zusammenstellung des Datensatzes manuell mit `as.factor(<Vektor>)` zu konvertieren.

```
> fac    <- as.factor(fac)
> myDf3 <- data.frame(fac, DV1, DV2, stringsAsFactors=FALSE)
> str(myDf3)
'data.frame': 3 obs. of 3 variables:
$ fac: Factor w/ 3 levels  "CG",...: 1 2 3
$ DV1: num  14 22 18
$ DV2: chr  "red" "blue" "blue"
```

Matrizen und Vektoren können mit `as.data.frame(<Objekt>)` in einen Datensatz umgewandelt werden. Dabei sind die in Matrizen und Vektoren notwendigerweise identischen Datentypen nachträglich zu konvertieren, wenn sie eigentlich unterschiedliche VariablenTypen repräsentieren. Dies ist z. B. dann der Fall, wenn numerische Vektoren und Zeichenketten in Matrizen zusammengefasst und so alle Elemente zu Zeichenketten gemacht wurden.

Listen können in Datensätze umgewandelt werden, wenn ihre Komponenten alle dieselbe Länge besitzen. Umgekehrt können Datensätze mit `data.matrix(<Datensatz>)` und auch `as.matrix(<Datensatz>)` zu Matrizen gemacht, wobei alle Werte in denselben Datentyp umgewandelt werden: bei `data.matrix()` immer in `numeric`, bei `as.matrix()` in den umfassendsten Datentyp, der notwendig ist, um alle Werte ohne Informationsverlust zu speichern (vgl. Abschn. 1.3.5). Bei der Umwandlung in eine Liste mit `as.list(<Datensatz>)` ist dies dagegen nicht notwendig.

3.2.2 Elemente auswählen und verändern

Um einzelne Elemente anzeigen zu lassen und diese zu ändern, lassen sich dieselben Befehle wie bei Listen verwenden.

```
> myDf1[[3]][2]                                # 2. Element der 3. Variable
[1] CG
Levels: CG T WL

> myDf1$rating                                # Variable rating
```

```
[1] 1 3 5 5 2 0 3 2 3 5 1 1

> myDf1$age[4]                                # Alter der 4. Person
[1] 34

> myDf1$IQ[10:12] <- c(99, 110, 89)    # ändere IQ-Werte für Pers 10-12
```

Als Besonderheit können Datensätze analog zu Matrizen mit dem Operator

- [*<Index Element der Komponente>*, *<Index der Komponente>*]

indiziert werden. Bei dieser Variante bleibt die gewohnte Reihenfolge von Zeile (entspricht einer Person) – Spalte (entspricht einer Variable) erhalten, ist also in vielen Fällen dem [[*<Index>*]] Operator vorzuziehen.

```
> myDf1[3, 4]                                # 3. Element der 4. Variable
[1] 25

> myDf1[4, "group"]                          # 4. Person, Variable group
[1] T
Levels: CG T WL
```

Wie bei Matrizen gilt das Weglassen eines Index unter Beibehaltung des Kommas als Anweisung, die Werte von allen Indizes der ausgelassenen Dimension anzuzeigen.⁶ Auch hier ist das Argument `drop=FALSE` notwendig, wenn ein einspältiges Ergebnis bei der Ausgabe weiterhin ein Datensatz sein soll. Bei der Arbeit mit Indexvektoren, um Spalten eines Datensatzes auszuwählen, ist häufig im voraus nicht absehbar, ob letztlich nur eine oder mehrere Spalten auszugeben sind. Um inkonsistentes Verhalten zu vermeiden, empfiehlt es sich in solchen Situationen, `drop=FALSE` in jedem Fall zu verwenden.

```
> myDf1[2, ]                                    # alle Werte der 2. Person
  id sex group age   IQ rating
2  2    m     CG   30  122      3

> myDf1[ , "age"]                            # alle Elemente Variable age
[1] 26 30 25 34 22 24 28 35 23 29 20 21

> myDf1[1:5, 4, drop=FALSE]                 # Variable als Datensatz
  age
1 26
2 30
3 25
4 34
5 22
```

Enthält der Datensatz möglicherweise fehlende Werte, sollten logische Indexvektoren mit `which()` in numerische Indizes konvertiert werden (vgl. Abschn. 2.2.2).

⁶Das Komma ist von Bedeutung: So würde etwa `<Datensatz>[3]` wie in Listen nicht einfach die dritte Variable von `<Datensatz>` zurückgeben, sondern einen Datensatz, dessen einzige Komponente diese Variable ist.

```
> (idxLog <- myDf1$sex == "f") # Indizes weibliche Personen
[1] TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE

> (idxNum <- which(idxLog)) # numerische Indizes
[1] 1 6 12

> myDf1[idxNum, ]
  id sex group age IQ rating
Z1  1   f      T  26 112      1
Z6  6   f     CG  24 113      0
Z12 12   f      T  21  98      1
```

3.2.3 Namen von Variablen und Beobachtungen

Die von arrays bekannte Funktion `dimnames(<Datensatz>)` dient dazu, die Namen eines Datensatzes auf beiden Dimensionen (Zeilen und Spalten, also meist Beobachtungsobjekte und Variablen) zu erfragen und auch zu ändern.⁷ Sie gibt eine Liste aus, in der die einzelnen Namen für jede der beiden Dimensionen als Komponenten enthalten sind. Wurden die Zeilen nicht benannt, werden ihre Nummern ausgegeben.

```
> dimnames(myDf1)
[[1]]
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12"

[[2]]
[1] "id" "sex" "group" "age" "IQ" "rating"
```

Die Namen der Variablen in einem Datensatz können mit der `names(<Datensatz>)` Funktion erfragt werden. Diese gibt nur die Variablennamen aus und ist damit gleichwertig zur Funktion `colnames(<Datensatz>)`, welche die Spaltennamen liefert.

```
> names(myDf1)
[1] "id" "sex" "group" "age" "IQ" "rating"
```

Variablennamen lassen sich verändern, indem der entsprechende Variablenname im obigen Ergebnisvektor ausgewählt und über eine Zuweisung umbenannt wird. Dabei kann entweder die Position direkt angegeben oder durch Vergleich mit dem gesuchten Variablennamen implizit ein logischer Indexvektor verwendet werden.

```
> names(myDf1)[3]
[1] "group"

> names(myDf1)[3] <- "fac"
> names(myDf1)
[1] "id" "sex" "fac" "age" "IQ" "rating"
```

⁷Namen werden als Attribut gespeichert und sind mit `attributes(<Datensatz>)` sichtbar (vgl. Abschn. 1.3).

```
> names(myDf1)[names(myDf1) == "fac"] <- "group"
> names(myDf1)
[1] "id" "sex" "group" "age" "IQ" "rating"
```

Die Bezeichnungen der Zeilen können mit `rownames(<Datensatz>)` bzw. `row.names(<Datensatz>)` ausgegeben und auch geändert werden. Der Vorgang ist analog zu jenem bei `names()`.

```
> (rows <- paste("Z", 1:12, sep=""))
[1] "Z1" "Z2" "Z3" "Z4" "Z5" "Z6" "Z7" "Z8" "Z9" "Z10" "Z11" "Z12"
```

```
> rownames(myDf1) <- rows
> head(myDf1)
  id sex group age IQ rating
Z1 1   f     T   26 112      1
Z2 2   m     CG  30 122      3
Z3 3   m     CG  25  95      5
Z4 4   m     T   34 102      5
Z5 5   m     WL  22  82      2
Z6 6   f     CG  24 113      0

> rownames(myDf1) <- NULL           # entferne Zeilennamen
```

3.2.4 Datensätze in den Suchpfad einfügen

Die in einem Datensatz vorhandenen Variablen sind außerhalb des Datensatzes unbekannt. Enthält ein Datensatz `myDf` die Variable `var`, so kann auf diese nur mit `myDf$var`, nicht aber einfach mit `var` zugegriffen werden. Nun kann es bequem sein, die Variablennamen auch ohne das wiederholte Aufführen von `myDf$` zu verwenden. Eine temporär wirkende Möglichkeit hierzu bietet `with()`.

```
> with(data=<Datensatz>, expr=<R-Befehle>)
```

Innerhalb der unter `expr` angegebenen Befehle sind die Variablennamen des unter `data` genannten Datensatzes bekannt. Der Datensatz selbst kann innerhalb von `with()` nicht verändert, sondern nur gelesen werden.

```
> with(myDf1, tapply(IQ, group, FUN=mean))
    KG      T      WL
105.50 101.00  85.25
```

Demselben Zweck dient in vielen Funktionen das Argument `data=<Datensatz>`, das es erlaubt, in anderen Argumenten der Funktion Variablen von `data` zu verwenden.

```
> xtabs(~ sex + group, data=myDf1)
      group
sex  CG  T  WL
  f    1  2   0
  m    3  2   4
```

Mit `attach(<Datensatz>)` ist es möglich, einen Datensatz in den Suchpfad einzuhängen und die Namen seiner Variablen so auch permanent ohne wiederholtes Voranstellen von `<Datensatz>$` verfügbar zu machen. Dies wird etwa an der Ausgabe von `search()` deutlich, die den Datensatz nach Einhängen in den Suchpfad mit aufführt (vgl. Abschn. 1.3.1).

```
> IQ[3]  
Fehler: Objekt "IQ" nicht gefunden  
  
> attach(myDf1)  
> IQ[3]  
[1] 95  
  
> search()[1:4]  
[1] ".GlobalEnv"  "myDf1"   "package:grDevices"  "package:datasets"
```

Wichtig ist, dass durch `attach()` im workspace Kopien aller Variablen des Datensatzes angelegt werden.⁸ Greift man daraufhin auf eine Variable `var` ohne Nennung von `myDf$` zu, so verwendet man diese Kopie. Insbesondere schlagen sich spätere Änderungen am Datensatz selbst nicht in den früher angelegten Kopien nieder, genauso wirken sich Veränderungen an den Kopien nicht auf den eigentlichen Datensatz aus. Datensätze sollten wegen dieser Gefahr, nicht synchronisierte Änderungen vorzunehmen, höchstens dann in den Suchpfad eingefügt werden, wenn alle Modifikationen an ihm abgeschlossen sind.

Mit dem Befehl `detach(<Datensatz>)` kann der Datensatz wieder aus dem Suchpfad entfernt werden, wenn nicht mehr auf seine Variablen zugegriffen werden muss. Dies sollte nicht vergessen werden, sonst besteht das Risiko, mit einem neuerlichen Aufruf von `attach()` denselben Datensatz mehrfach verfügbar zu machen, was für Verwirrung sorgen kann.

```
> IQ[3] <- 130                                # Änderung der Kopie  
> IQ[3]  
[1] 130  
  
> myDf1$IQ[3]                                 # Original  
[1] 95  
  
> detach(myDf1)  
> IQ  
Fehler: Objekt "IQ" nicht gefunden
```

3.3 Datensätze transformieren

Bevor Datensätze analysiert werden können, müssen sie häufig in eine andere als ihre ursprüngliche Form gebracht werden – etwa um neue Variablen zu bilden, Fälle bzw. Variablen auszuwählen

⁸Bei sehr großen Datensätzen empfiehlt es sich daher aus Gründen der Speichernutzung, nur eine geeignete Teilmenge von Fällen mit `attach()` verfügbar zu machen (vgl. Abschn. 3.3.3).

sowie Datensätze zu teilen bzw. unterschiedliche Datensätze zusammenzuführen.⁹

3.3.1 Variablen hinzufügen und entfernen

Wie bei Listen (vgl. Abschn. 3.1.1) können einem bestehenden Datensatz neue Variablen mit den Operatoren `<Datensatz>$neue Variable` und `<Datensatz>["neue Variable"]` hinzugefügt werden. Analog zum Vorgehen bei Matrizen kann an einen Datensatz auch mit `cbind(<Datensatz>, <Vektor>, ...)` eine weitere Variable als Spalte angehängt werden, sofern der zugehörige Vektor passend viele Elemente umfasst.¹⁰

Im Beispiel soll der Beziehungsstatus der Personen dem Datensatz hinzugefügt werden.

```
> married <- sample(c(TRUE, FALSE), nrow(myDf1), replace=TRUE)
> myDf2 <- myDf1                                # erstelle Kopie
> myDf2$married1 <- married                      # Möglichkeit 1
> myDf2["married2"] <- married                  # Möglichkeit 2
> myDf3 <- cbind(myDf1, married)                # Möglichkeit 3
> head(myDf3, n=3)
  id sex group age  IQ rating married
1 1   f     T  26 112      1    TRUE
2 2   m    CG  30 122      3    TRUE
3 3   m    CG  25  95      5   FALSE
```

Alternativ kann auch `transform()` Verwendung finden.

```
> transform(<Datensatz>, <Variablenname>=<Ausdruck>)
```

Unter `<Ausdruck>` ist anzugeben, wie sich die Werte der neuen Variable ergeben, die unter `<Variablenname>` gespeichert und an `<Datensatz>` angehängt wird. Wie im Aufruf von `data.frame()` werden Variablen mit Zeichenketten dabei durch die Transformation automatisch in Faktoren konvertiert, wenn nicht `stringsAsFactors=FALSE` gesetzt ist. Im Beispiel soll das Quadrat des Ratings angefügt werden. Die Variablennamen des Datensatzes sind innerhalb von `transform()` bekannt.

```
> myDf4 <- transform(myDf1, rSq=rating^2)
> head(myDf4, n=3)
  id sex group age  IQ rating rSq
1 1   f     T  26 112      1     1
2 2   m    CG  30 122      3     9
3 3   m    CG  25  95      5    25
```

⁹Das Paket `dplyr` (Wickham & Francois, 2014) enthält spezialisierte Funktionen, die diese Arbeitsschritte systematisieren und besonders bequem durchführbar machen.

¹⁰Dagegen ist das Ergebnis von `cbind(<Vektor1>, <Vektor2>)` eine Matrix. Dies ist insbesondere wichtig, wenn numerische Daten und Zeichenketten zusammengefügt werden – in einer Matrix würden die numerischen Werte automatisch in Zeichenketten konvertiert.

Variablen eines Datensatzes werden gelöscht, indem ihnen die leere Menge `NULL` zugewiesen wird – im Fall mehrerer Variablen gleichzeitig in Form einer Liste mit der Komponente `NULL`.¹¹

```
> dfTemp      <- myDf1                      # Kopie erstellen
> dfTemp$group <- NULL                     # eine Variable löschen
> head(dfTemp, n=3)
  id sex age  IQ rating
1 1   w  26 112     1
2 2   m  30 122     3
3 3   m  25  95     5

> delVars <- c("sex", "IQ")                 # mehrere Variablen löschen
> dfTemp[delVars] <- list(NULL)
> head(dfTemp, n=3)
  id age rating
Z1 1   26     1
Z2 2   30     3
Z3 3   25     5
```

3.3.2 Datensätze sortieren

Datensätze werden ebenso wie Matrizen mit `order()` sortiert (vgl. Abschn. 2.8.6).

```
> order(<Vektor>, na.last=TRUE, decreasing=FALSE)
```

Unter `<Vektor>` ist die Variable (Spalte) eines Datensatzes anzugeben, die in eine Reihenfolge zu bringen ist. `na.last` ist per Voreinstellung auf `TRUE` gesetzt und sorgt ggf. dafür, dass Indizes fehlender Werte zum Schluss ausgegeben werden. Die Voreinstellung `decreasing=FALSE` bewirkt eine aufsteigende Reihenfolge. Zeichenketten werden in alphabetischer Reihenfolge sortiert. Die Reihenfolge bei Faktoren wird dagegen von der Reihenfolge der Stufen bestimmt, die nicht deren alphabetischer Reihenfolge entsprechen muss (vgl. Abschn. 2.6.5).

`order()` gibt einen Indexvektor aus, der die Zeilenindizes des Datensatzes in der Reihenfolge der zu ordnenden Variablenwerte enthält. Soll der gesamte Datensatz in der Reihenfolge dieser Variable angezeigt werden, ist der ausgegebene Indexvektor zum Indizieren der Zeilen des Datensatzes zu benutzen.

```
> (idx1 <- order(myDf1$rating))          # sortiere nach rating
[1] 6 1 11 12 5 8 2 7 9 3 4 10

> head(myDf1[idx1, ])                  # erste Zeilen des sortierten Datensatzes
  id sex group age  IQ rating
6   6   f    CG  24 113     0
1   1   f     T  26 112     1
11 11   m    CG  20  92     1
```

¹¹Das genannte Vorgehen wirft die Frage auf, wie sich allen Elementen einer Variable gleichzeitig der Wert `NULL` zuweisen lässt, statt die Variable zu löschen. Dies ist durch `<Datensatz>$<Variable> <- list(NULL)` möglich.

```
12 12      f      T   21   98       1
 5   5      m     WL   22   82       2
 8   8      m     WL   35   90       2
```

Soll nach zwei Kriterien sortiert werden, weil die Reihenfolge durch eine Variable noch nicht vollständig festgelegt ist, können weitere Datenvektoren in der Rolle von Sortierkriterien als Argumente für `order()` angegeben werden.

```
# sortiere myDf1 primär nach group und innerhalb jeder Gruppe nach IQ
> (idx2 <- order(myDf1$group, myDf1$IQ))
[1] 11 3 6 2 7 12 4 1 10 5 9 8
```

```
> head(myDf1[idx2, ])      # erste Zeilen des sortierten Datensatzes
  id sex group age  IQ rating
11 11    m    CG  20  92      1
 3   3    m    CG  25  95      5
 6   6    f    CG  24 113      0
 2   2    m    CG  30 122      3
 7   7    m      T  28  92      3
12 12    f      T  21  98      1
```

3.3.3 Teilmengen von Daten mit `subset()` auswählen

Bei der Analyse eines Datensatzes möchte man häufig eine Auswahl der Daten treffen, etwa nur Personen aus einer bestimmten Gruppe untersuchen, nur Beobachtungsobjekte berücksichtigen, die einen bestimmten Testwert überschreiten, oder aber die Auswertung auf eine Teilgruppe von Variablen beschränken. Die Auswahl von Beobachtungsobjekten auf der einen und Variablen auf der anderen Seite unterscheidet sich dabei konzeptuell nicht voneinander, vielmehr kommen in beiden Fällen die bereits bekannten Strategien zur Indizierung von Objekten zum Tragen.

Eine übersichtlichere Methode zur Auswahl von Fällen und Variablen als die in Abschn. 3.2.2 vorgestellte Indizierung bietet die Funktion `subset()`. Sie gibt einen passenden Datensatz zurück und eignet sich damit in Situationen, in denen eine Auswertungsfunktion einen Datensatz als Argument erwartet – etwa wenn ein inferenzstatistischer Test nur mit einem Teil der Daten durchgeführt werden soll.

```
> subset(x=<Datensatz>, subset=<Indexvektor>, select=<Spalten>)
```

Das Argument `subset` ist ein Indexvektor zum Indizieren der Zeilen, der sich häufig aus einem logischen Vergleich der Werte einer Variable mit einem Kriterium ergibt.¹² Analog dient das Argument `select` dazu, eine Teilmenge der Spalten auszuwählen, wofür ein Vektor mit auszugebenden Spaltenindizes oder -namen benötigt wird. Dabei müssen Variablennamen ausnahmsweise nicht in Anführungszeichen stehen. Fehlen konkrete Indexvektoren für `subset` oder `select`, werden alle Zeilen bzw. Spalten ausgegeben. Innerhalb von `subset()` sind die Variablennamen des Datensatzes bekannt, ihnen muss also im Ausdruck zur Bildung eines Indexvektors nicht `<Datensatz>$` vorangestellt werden.

¹²Da fehlende Werte innerhalb von `subset` als `FALSE` behandelt werden, ist es hier nicht notwendig, logische Indizes mit `which()` in numerische umzuwandeln.

```
> subset(myDf1, select=c(group, IQ))           # Variablen group und IQ
   group  IQ
1      T 112
2     CG 122
3     CG  95
4      T 102
5     WL  82
6     CG 113
7      T  92
8     WL  90
9     WL  88
10    WL  81
11    CG  92
12    T  98

> subset(myDf1, sex == "f")                  # Daten weibliche Personen
   id sex group age  IQ rating
1  1   f     T  26 112      1
6  6   f     CG  24 113      0
12 12  f     T  21  98      1

> subset(myDf1, id == rating)               # Wert id = Wert rating
   id sex group age  IQ rating
1  1   f     T  26 112      1
```

Um alle bis auf einige Variablen auszugeben, eignet sich ein negatives Vorzeichen vor dem numerischen Index oder vor dem Namen der wegzulassenden Variablen.

```
> subset(myDf1, sex == "f", select=-sex)
   id group age  IQ rating
1  1     T  26 112      1
6  6    CG  24 113      0
12 12    T  21  98      1
```

Alle Variablen, deren Namen einem bestimmten Muster entsprechen, können mit den in Abschn. 2.12.4 vorgestellten Funktionen zur Suche nach Zeichenfolgen ausgewählt werden.

```
> dfTemp <- myDf1                      # kopiere myDf1

# neue Variablennamen, so dass sich zwei Gruppen von Variablen ergeben
> (names(dfTemp) <- paste(rep(c("A", "B"), each=3), 100:102, sep=""))
[1] "A100" "A101" "A102" "B100" "B101" "B102"

# Index der mit B beginnende Variablen
> (colIdx <- grep("^B.*$", names(dfTemp)))
[1] 4 5 6

> subset(dfTemp, B100 > 25, select=colIdx)
```

	B100	B101	B102
1	26	112	1
2	30	122	3
4	34	102	5
7	28	92	3
8	35	90	2
10	29	81	5

Mit `subset()` ist auch die Auswahl nach mehreren Kriterien gleichzeitig möglich, indem $\langle \rightarrow \text{Indexvektor} \rangle$ durch entsprechend erweiterte logische Ausdrücke gebildet wird. Dabei kann es der Fall sein, dass mehrere Bedingungen gleichzeitig erfüllt sein müssen (logisches UND, `&`), oder es ausreicht, wenn bereits eines von mehreren Kriterien erfüllt ist (logisches ODER, `|`).

```
# alle männlichen Personen mit einem Rating größer als 2
> subset(myDf1, (sex == "m") & (rating > 2))
   id sex group age  IQ rating
2   2   m    CG  30 122      3
3   3   m    CG  25  95      5
4   4   m     T  34 102      5
7   7   m     T  28  92      3
9   9   m    WL  23  88      3
10 10   m    WL  29  81      5

# alle Personen mit einem eher hohen ODER eher niedrigen IQ-Wert
> subset(myDf1, (IQ < 90) | (IQ > 110))
   id sex group age  IQ rating
1   1   f     T  26 112      1
2   2   m    CG  30 122      3
5   5   m    WL  22  82      2
6   6   f    CG  24 113      0
9   9   m    WL  23  88      3
10 10   m    WL  29  81      5
```

Für die Auswahl von Fällen, deren Wert auf einer Variable aus einer Menge bestimmter Werte stammen soll (logisches ODER), gibt es eine weitere Möglichkeit: Mit dem Operator $\langle \text{Menge1} \rangle \rightarrow \%in\% \langle \text{Menge2} \rangle$ als Kurzform von `is.element()` kann ebenfalls ein logischer Indexvektor zur Verwendung in `subset()` gebildet werden. Dabei prüft $\langle \text{Menge1} \rangle \%in\% \langle \text{Menge2} \rangle$ für jedes Element von $\langle \text{Menge1} \rangle$, ob es auch in $\langle \text{Menge2} \rangle$ vorhanden ist und gibt einen logischen Vektor mit den einzelnen Ergebnissen aus.

```
# Personen aus Wartelisten- ODER Kontrollgruppe
> subset(myDf1, group %in% c("CG", "WL"))
   id sex group age  IQ rating
2   2   m    CG  30 122      3
3   3   m    CG  25  95      5
5   5   m    WL  22  82      2
6   6   f    CG  24 113      0
8   8   m    WL  35  90      2
```

```

9   9   m    WL  23  88      3
10 10   m    WL  29  81      5
11 11   m    CG  20  92      1

```

3.3.4 Doppelte und fehlende Werte behandeln

Doppelte Werte können in Datensätzen etwa auftreten, nachdem sich teilweise überschneidende Daten aus unterschiedlichen Quellen in einem Datensatz integriert wurden. Alle später auftretenden Duplikate mehrfach vorhandener Zeilen werden durch `duplicated(<Datensatz>)` identifiziert und durch `unique(<Datensatz>)` ausgeschlossen (vgl. Abschn. 2.3.1). Lediglich die jeweils erste Ausfertigung bleibt so erhalten.

```

# Datensatz mit doppelten Werten herstellen
> myDfDouble <- rbind(myDf1, myDf1[sample(1:nrow(myDf1), 4), ])

# doppelte Zeilen identifizieren (alle Ausfertigungen)
> duplicated(myDfDouble) | duplicated(myDfDouble, fromLast=TRUE)
[1] FALSE FALSE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
[12] FALSE TRUE TRUE TRUE TRUE

> myDfUnique <- unique(myDfDouble)      # doppelte Zeilen ausschließen
> any(duplicated(myDfUnique))           # Kontrolle: noch doppelte?
[1] FALSE

```

Fehlende Werte werden in Datensätzen weitgehend wie in Matrizen behandelt (vgl. Abschn. 2.11.4). Auch hier muss also `is.na()` benutzt werden, um das Vorhandensein fehlender Werte zu prüfen.

```

> myDfNA          <- myDf1
> myDfNA$IQ[4]    <- NA
> myDfNA$rating[5] <- NA
> is.na(myDfNA)[1:5, c("age", "IQ", "rating")]
  age     IQ rating
1 FALSE  FALSE FALSE
2 FALSE  FALSE FALSE
3 FALSE  FALSE FALSE
4 FALSE  TRUE  FALSE
5 FALSE  FALSE  TRUE

# prüfe jede Variable, ob sie mindestens ein NA enthält
> apply(is.na(myDfNA), 2, any)
  id   sex group   age   IQ rating
FALSE FALSE FALSE FALSE TRUE  TRUE

```

Eine weitere Funktion zur Behandlung fehlender Werte ist `complete.cases(<Datensatz>)`. Sie liefert einen logischen Indexvektor zurück, der für jedes Beobachtungsobjekt (jede Zeile) angibt, ob fehlende Werte vorliegen. Die vollständigen Fälle, oder die Fälle mit fehlenden Werten

können mit `subset()` ausgegeben werden. Für die Zeilen mit fehlenden Werten ist dabei der Indexvektor aus der logischen Negation des Ergebnisses von `complete.cases()` zu bilden.

```
> complete.cases(myDfNA)
[1] TRUE TRUE TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

> subset(myDfNA, !complete.cases(myDfNA))      # Zeilen mit fehlenden Werten
   id sex group age IQ rating
4  4   m     T   34 NA     5
5  5   m    WL   22  82    NA
```

Weiterhin können wie bei Matrizen alle Zeilen mit `na.omit()` gelöscht werden, in denen Werte fehlen.

```
> head(na.omit(myDfNA))                         # nur vollständige Zeilen
   id sex group age IQ rating
1  1   f     T   26 112     1
2  2   m    CG   30 122     3
3  3   m    CG   25  95     5
6  6   f    CG   24 113     0
7  7   m     T   28  92     3
8  8   m    WL   35  90     2
```

3.3.5 Datensätze teilen

Wenn die Beobachtungen (Zeilen) eines Datensatzes in durch die Stufen eines Faktors festgelegte Gruppen aufgeteilt werden sollen, kann dies mit `split()` geschehen.

```
> split(x=<Datensatz>, f=<Faktor>)
```

Für jede Zeile des Datensatzes `x` muss der Faktor `f` die Gruppenzugehörigkeit angeben und deshalb die Länge `nrow(x)` besitzen. Auch wenn `f` Teil des Datensatzes ist, muss der Faktor vollständig mit `<Datensatz>$<Faktor>` angegeben werden. Sollen die Zeilen des Datensatzes in Gruppen aufgeteilt werden, die sich aus der Kombination der Stufen zweier Faktoren ergeben, sind diese wie bei `tapply()` in eine Liste einzuschließen (vgl. Abschn. 2.7.10). In diesem Fall sorgt das Argument `drop=TRUE` dafür, dass nur jene Stufen-Kombinationen berücksichtigt werden, für die auch Beobachtungen vorhanden sind.

Das Ergebnis ist eine Liste, die für jede sich aus den Stufen von `f` ergebende Gruppe eine Komponente in Form eines Datensatzes besitzt. Diese Datensätze können etwa dazu dienen, Auswertungen getrennt nach Gruppen vorzunehmen (vgl. Abschn. 3.4.3).

```
> split(myDf1, myDf1$group)
$CG
   id sex group age IQ rating
2  2   m    CG   30 122     3
3  3   m    CG   25  95     5
6  6   f    CG   24 113     0
11 11  m    CG   20  92     1
```

```
$T
  id sex group age IQ rating
1 1 f T 26 112 1
4 4 m T 34 102 5
7 7 m T 28 92 3
12 12 f T 21 98 1

$WL
  id sex group age IQ rating
5 5 m WL 22 82 2
8 8 m WL 35 90 2
9 9 m WL 23 88 3
10 10 m WL 29 81 5

# teile Beobachtungen nach sex und group auf
> split(myDf1, list(myDf1$sex, myDf1$group)) # ...
```

3.3.6 Datensätze zeilen- oder spaltenweise verbinden

Wenn zwei oder mehr Datensätze df vorliegen, die hinsichtlich ihrer Variablen identisch sind, so fügt die Funktion `rbind(df1, df2, ...)` die Datensätze analog zum Vorgehen bei Matrizen zusammen, indem sie sie untereinander anordnet. Auf diese Weise könnten z. B. die Daten mehrerer Teilstichproben kombiniert werden, an denen dieselben Variablen erhoben wurden.¹³ Wenn die Datensätze Gruppierungsfaktoren enthalten, ist zunächst sicherzustellen, dass alle dieselben Faktorstufen in derselben Reihenfolge besitzen (vgl. Abschn. 2.6.3, 2.6.5).

```
> (dfNew <- data.frame(id=13:15, group=c("CG", "WL", "T"),
+                         sex=c("f", "f", "m"), age=c(18, 31, 21),
+                         IQ=c(116, 101, 99), rating=c(4, 4, 1)))
  id sex group age IQ rating
1 13 f CG 18 116 4
2 14 f WL 31 101 4
3 15 m T 21 99 1

> dfComb <- rbind(myDf1, dfNew)
> dfComb[11:15, ]
  id sex group age IQ rating
11 11 m CG 20 92 1
12 12 f T 21 98 1
13 13 f CG 18 116 4
14 14 f WL 31 101 4
15 15 m T 21 99 1
```

¹³`rbind_all()` aus dem Paket `dplyr` kann auch Datensätze miteinander verbinden, die sich bzgl. der Variablen unterscheiden. Teilweise nicht vorhandene Beobachtungen für Variablen werden dafür auf `NA` gesetzt.

Beim Zusammenfügen mehrerer Datensätze besteht die Gefahr, Fälle doppelt aufzunehmen, wenn es Überschneidungen hinsichtlich der Beobachtungsobjekte gibt. `duplicated()` und `unique()` eignen sich dazu, eindeutige bzw. mehrfach vorkommende Zeilen eines Datensatzes zu identifizieren (vgl. Abschn. 3.3.4).

Liegen von denselben Beobachtungsobjekten zwei Datensätze `df1` und `df2` aus unterschiedlichen Variablen vor, können diese analog zum Anhängen einzelner Variablen an einen Datensatz mit `cbind(df1, df2)` oder `data.frame(df1, df2)` so kombiniert werden, dass die Variablen nebeneinander angeordnet sind. Dabei ist sicherzustellen, dass die zu demselben Beobachtungsobjekt gehörenden Daten in derselben Zeile jedes Datensatzes stehen.

3.3.7 Datensätze mit `merge()` zusammenführen

Um flexibel zwei Datensätze zusammenzuführen, die sich nur teilweise in den Variablen oder in den Beobachtungsobjekten entsprechen, kann `merge()` zum Einsatz kommen.

```
> merge(x=<Datensatz1>, y=<Datensatz2>, by.x, by.y, by, all.x, all.y, all)
```

Zunächst sei die Situation betrachtet, dass die unter `x` und `y` angegebenen Datensätze Daten derselben Beobachtungsobjekte beinhalten, die über eine eindeutige ID identifiziert sind. Dabei sollen einige Variablen bereits in beiden, andere Variablen hingegen nur in jeweils einem der beiden Datensätze vorhanden sein. Die in beiden Datensätzen gleichzeitig enthaltenen Variablen enthalten dann dieselbe Information, da die Daten von denselben Beobachtungsobjekten stammen. Ohne weitere Argumente ist das Ergebnis von `merge()` ein Datensatz, der jede der in `x` und `y` vorkommenden Variablen nur einmal enthält, identische Spalten werden also nur einmal aufgenommen.¹⁴

Beispiel sei ein Datensatz mit je zwei Messungen eines Merkmals an drei Personen mit eindeutiger ID. In einem zweiten Datensatz sind für jede ID die Informationen festgehalten, die konstant über die Messwiederholungen sind – hier das Geschlecht und eine Gruppenzugehörigkeit.¹⁵

```
# Datensatz mit 2 Messwerten pro ID
> (IDDV <- data.frame(ID=factor(rep(1:3, each=2)),
+                         DV=round(rnorm(6, mean=100, sd=15))))
   ID  DV
1  1  93
2  1 105
3  2 112
4  2 101
5  3 109
6  3 101

# Datensatz mit Informationen, die pro ID konstant sind
```

¹⁴Zur Identifizierung gleicher Variablen werden die Spaltennamen mittels `intersect(names(x), names(y))` herangezogen. Bei Gruppierungsfaktoren ist es wichtig, dass sie in beiden Datensätzen dieselben Stufen in derselben Reihenfolge besitzen.

¹⁵Diese Datenstruktur entspricht einer *normalisierten* Datenbank mit mehreren tables zur Vermeidung von Redundanzen.

```
> (IV <- data.frame(ID=factor(1:3),
+                     IV=factor(c("A", "B", "A")),
+                     sex=factor(c("f", "f", "m"))))

  ID IV sex
1  1  A  f
2  2  B  f
3  3  A  m
```

Als Ziel soll ein Datensatz erstellt werden, der die pro Person konstanten und veränderlichen Variablen integriert, wobei die Zuordnung von Informationen über die ID geschieht. Die pro Person über die Messwiederholungen konstanten Werte werden dabei von `merge()` automatisch passend oft wiederholt.

```
> merge(IDDV, IV)
  ID DV IV sex
1  1  93  A  f
2  1 105  A  f
3  2 112  B  f
4  2 101  B  f
5  3 109  A  m
6  3 101  A  m
```

Verfügen `x` und `y` im Prinzip über dieselben Variablen, jedoch mit abweichender Bezeichnung, kann über die Argumente `by.x` und `by.y` manuell festgelegt werden, welche ihrer Variablen übereinstimmen und deshalb nur einmal aufgenommen werden sollen. Als Wert für `by.x` und `by.y` muss jeweils ein Vektor mit Namen oder Indizes der übereinstimmenden Spalten eingesetzt werden. Alternativ kann dies auch ein logischer Vektor sein, der für jede Spalte von `x` (im Fall von `by.x`) bzw. jede Spalte von `y` (im Fall von `by.y`) angibt, ob sie eine auch im jeweils anderen Datensatz vorkommende Variable darstellt. Beide Argumente `by.x` und `by.y` müssen gleichzeitig verwendet werden und müssen dieselbe Anzahl von gleichen Spalten bezeichnen. Haben beide Datensätze insgesamt dieselbe Anzahl von Spalten, kann auch auf das Argument `by` zurückgegriffen werden, das sich dann auf die Spalten beider Datensätze gleichzeitig bezieht.

Im folgenden Beispiel soll die Spalte der Initialen künstlich als nicht übereinstimmende Variable gekennzeichnet werden, indem das zweite Element des an `by.x` und `by.y` übergebenen Vektors auf `FALSE` gesetzt wird.

```
> (dfa <- data.frame(ID=1:4, initials=c("AB", "CD", "EF", "GH"),
+                      IV1=c("-", "-", "+", "+"), DV1=c(10, 19, 11, 14)))
  ID initials IV1 DV1
1  1        AB  -  10
2  2        CD  -  19
3  3        EF  +  11
4  4        GH  +  14

> (dfb <- data.frame(ID=1:4, initials=c("AB", "CD", "EF", "GH"),
+                      IV2=c("A", "B", "A", "B"), DV2=c(91, 89, 92, 79)))
  ID initials IV2 DV2
```

1	1	AB	A	91
2	2	CD	B	89
3	3	EF	A	92
4	4	GH	B	79

```
> merge(dfA, dfB, by.x=c(TRUE, FALSE, FALSE, FALSE),
+           by.y=c(TRUE, FALSE, FALSE, FALSE))
   ID initials.x IV1 DV1 initials.y IV2 DV2
1  1         AB   -   10         AB   A   91
2  2         CD   -   19         CD   B   89
3  3         EF   +   11         EF   A   92
4  4         GH   +   14         GH   B   79
```

Berücksichtigt werden bei dieser Art des Zusammenfügens nur Zeilen, bei denen die in beiden Datensätzen vorkommenden Variablen identische Werte aufweisen – im Kontext von Datenbanken wird dieses Verhalten als *inner join* bezeichnet. Dabei werden alle Zeilen entfernt, deren Werte für die gemeinsamen Variablen zwischen den Datensätzen abweichen. Werden mit `by` Spalten als übereinstimmend gekennzeichnet, für die tatsächlich aber keine Zeile identische Werte aufweist, ist das Ergebnis deshalb ein leerer Datensatz.

```
> (dfC <- data.frame(ID=3:6, initials=c("EF", "GH", "IJ", "KL"),
+                      IV2=c("A", "B", "A", "B"), DV2=c(92, 79, 101, 81)))
   ID initials IV2 DV2
1  3         EF   A   92
2  4         GH   B   79
3  5         IJ   A  101
4  6         KL   B  81

> merge(dfA, dfC)
   ID initials IV1 DV1 IV2 DV2
1  3         EF   +   11   A   92
2  4         GH   +   14   B   79
```

Um das Weglassen solcher Zeilen zu verhindern, können die Argumente `all.x` bzw. `all.y` auf `TRUE` gesetzt werden. `all.x` bewirkt dann, dass alle Zeilen in `x`, die auf den übereinstimmenden Variablen andere Werte als in `y` haben, ins Ergebnis aufgenommen werden (im Kontext von Datenbanken ein *left outer join*). Die in `y` (aber nicht in `x`) enthaltenen Variablen werden für diese Zeilen auf `NA` gesetzt. Für das Argument `all.y` gilt dies analog (bei Datenbanken ein *right outer join*). Das Argument `all=TRUE` steht kurz für `all.x=TRUE` in Kombination mit `all.y=TRUE` (bei Datenbanken ein *full outer join*).

Im Beispiel sind die Werte bzgl. der übereinstimmenden Variablen in den ersten beiden Zeilen von `dfC` identisch mit jenen in `dfA`. Darüber hinaus enthält `dfC` jedoch auch zwei Zeilen mit Werten, die sich auf den übereinstimmenden Variablen von jenen in `dfA` unterscheiden. Um diese Zeilen im Ergebnis von `merge()` einzuschließen, muss deshalb `all.y=TRUE` gesetzt werden.

```
> merge(dfA, dfC, all.y=TRUE)
   ID initials IV1 DV1 IV2 DV2
```

		EF	+	11	A	92
2	4	GH	+	14	B	79
3	5	IJ	<NA>	NA	A	101
4	6	KL	<NA>	NA	B	81

Analoges gilt für das Einschließen der ersten beiden Zeilen von `dfA`. Diese beiden Zeilen haben andere Werte auf den übereinstimmenden Variablen als die Zeilen in `dfC`. Damit sie im Ergebnis auftauchen, muss `all.x=TRUE` gesetzt werden.

```
> merge(dfA, dfC, all.x=TRUE, all.y=TRUE)
   ID initials IV1 DV1 IV2 DV2
1  1      AB   -  10 <NA>  NA
2  2      CD   -  19 <NA>  NA
3  3      EF   +  11    A  92
4  4      GH   +  14    B  79
5  5      IJ <NA>  NA    A 101
6  6      KL <NA>  NA    B  81
```

3.3.8 Organisationsform einfacher Datensätze ändern

Bisweilen weicht der Aufbau eines Datensatzes von dem beschriebenen ab, etwa wenn sich Daten derselben, in verschiedenen Bedingungen erhobene Variablen (AVn) in unterschiedlichen Spalten befinden, die mit den Bedingungen korrespondieren. Wurde in jeder Bedingung eine andere Menge von Personen beobachtet, enthält eine Zeile dann nicht mehr die Daten einer einzelnen Person. Um diese Organisationsform in die übliche zu überführen, eignet sich `stack()`: Ziel ist es also, eine Zeile pro Person, eine Spalte für die Werte der AV in allen Bedingungen und eine Spalte für die zu jedem Wert gehörende Stufe des Gruppierungsfaktors (UV) zu erhalten.

```
> stack(x=<Liste>, select=<Spalten>)
```

Unter `x` ist eine Liste mit benannten Komponenten (meist ein Datensatz) anzugeben, deren zu reorganisierende Variablen über das Argument `select` bestimmt werden. Das Argument erwartet einen Vektor mit Spaltenindizes oder Variablennamen. Das Ergebnis ist ein Datensatz, in dessen erster Variable `values` sich alle Werte der AV befinden. Diese Variable entsteht, indem die ursprünglichen Spalten wie durch den Befehl `c(<Spalte1>, <Spalte2>, ...)` aneinander gehängt werden. Die zweite Variable `ind` ist ein Faktor, dessen Stufen codieren, aus welcher Spalte von `x` ein einzelner Wert der Variable `values` stammt. Hierzu werden die Variablennamen von `x` herangezogen.

```
> Nj      <- 4
> vec1    <- sample(1:10, Nj, replace=TRUE)
> vec2    <- sample(1:10, Nj, replace=TRUE)
> vec3    <- sample(1:10, Nj, replace=TRUE)
> dfTemp <- data.frame(cond1=vec1, cond2=vec2, cond3=vec3)
> (res    <- stack(dfTemp, select=c("cond1", "cond3")))
  values    ind
1       3  cond1
```

```

2      2  cond1
3      4  cond1
4      3  cond1
5      6  cond3
6      8  cond3
7      7  cond3
8      3  cond3

> str(res)
'data.frame': 8 obs. of 2 variables:
$ values: int 3 2 4 3 6 8 7 3
$ ind    : Factor w/ 2 levels "cond1","cond3": 1 1 1 1 2 2 2 2

```

Das Ergebnis von `stack()` wird durch `unstack()` umgekehrt. Diese Funktion transformiert also einen Datensatz, der aus einer Variable mit den Werten der AV und einer Variable mit den zugehörigen Faktorstufen besteht, in einen Datensatz mit so vielen Spalten wie Faktorstufen. Dabei beinhaltet jede Spalte die zu einer Stufe gehörenden Werte der AV. Das Ergebnis von `unstack()` ist nur dann ein Datensatz, wenn alle Faktorstufen gleich häufig vorkommen, wenn also die resultierenden Variablen dieselbe Länge aufweisen. Andernfalls ist das Ergebnis eine Liste.

```

> unstack(x=<Datensatz>, form=<Modellformel>)

> unstack(res)
  cond1  cond3
1      3      6
2      2      8
3      4      7
4      3      3

```

Kommen im Datensatz x mehrere AVn und UVn vor, so kann über eine an das Argument `form` zu übergebende *Modellformel* $\text{AV} \sim \text{UV}$ festgelegt werden, welche AV ausgewählt und nach welcher UV die Trennung der Werte der AV vorgenommen werden soll (vgl. Abschn. 5.2).

```

# füge zwei neue Variablen zum Datensatz res hinzu
> res$IVnew <- factor(sample(rep(c("A", "B"), Nj), 2*Nj, replace=FALSE))
> res$DVnew <- sample(100:200, 2*Nj)
> unstack(res, DVnew ~ IVnew)
   A     B
1 183  193
2 129  115
3 142  140
4 194  134

```

Die Organisationsformen eines Datensatzes, zwischen denen `stack()` und `unstack()` wechseln, werden im Kontext von Daten aus Messwiederholungen als *Long-Format* und *Wide-Format* bezeichnet. Häufig sind die Datensätze dann jedoch zu komplex, um noch mit diesen Funktionen bearbeitet werden zu können. Für solche Situationen ist die im folgenden Abschnitt beschriebene Funktion `reshape()` geeignet.

3.3.9 Organisationsform komplexer Datensätze ändern

Wurden an denselben Beobachtungsobjekten zu verschiedenen Messzeitpunkten Daten derselben AV erhoben, können die Werte jeweils eines Messzeitpunkts als zu einer eigenen Variable gehörend betrachtet werden. In einem Datensatz findet sich jede dieser Variablen dann in einer separaten Spalte. Diese Organisationsform folgt dem bisher dargestellten Prinzip, dass pro Zeile die Daten jeweils eines Objekts aus verschiedenen Variablen stehen. Eine solche Strukturierung wird als Wide-Format bezeichnet, weil der Datensatz durch mehr Messzeitpunkte an Spalten gewinnt, also breiter wird. Das Wide-Format entspricht einer multivariaten Betrachtungsweise von Daten aus Messwiederholungen (vgl. Abschn. 7.4.3, 12.6.4). Es setzt voraus, dass die Objekte zu denselben Zeitpunkten beobachtet wurden und damit der Messzeitpunkt pro Spalte konstant ist.

Für die univariat formulierte Analyse von abhängigen Daten (vgl. Abschn. 7.4) ist jedoch oft die Organisation im Long-Format notwendig. Die zu den verschiedenen Messzeitpunkten gehörenden Werte eines Beobachtungsobjekts stehen hier in separaten Zeilen. Auf diese Weise beinhalten mehrere Zeilen Daten desselben Beobachtungsobjekts. Der Name des Long-Formats leitet sich daraus ab, dass mehr Messzeitpunkte zu mehr Zeilen führen, der Datensatz also länger wird. Wichtig bei Verwendung dieses Formats ist zum einen das Vorhandensein eines Faktors, der codiert, von welchem Objekt eine Beobachtung stammt. Diese Variable ist dann jeweils über so viele Zeilen konstant, wie es Messzeitpunkte gibt. Zum anderen muss ein Faktor existieren, der den Messzeitpunkt codiert. Das Long-Format eignet sich auch für Situationen, in denen mehrere Objekte zu verschiedenen Zeitpunkten unterschiedlich häufig beobachtet wurden.

Vorgehen bei einem Messwiederholungsfaktor

Im Beispiel sei an vier Personen eine AV zu drei Messzeitpunkten erhoben worden. Bei zwei der Personen sei dies in Bedingung A, bei den anderen beiden in Bedingung B einer Manipulation geschehen. Damit liegen zwei UVn vor, zum einen als Intra-Gruppen Faktor der Messzeitpunkt, zum anderen ein Zwischen-Gruppen Faktor (*Split-Plot Design*, vgl. Abschn. 7.7). Zunächst soll demonstriert werden, wie sich das Long-Format manuell aus gegebenen Vektoren herstellen lässt. Soll mit einem solchen Datensatz etwa eine univariate Varianzanalyse mit Messwiederholung gerechnet werden, muss sowohl die Personen- bzw. Blockzugehörigkeit eines Messwertes als auch der Messzeitpunkt jeweils in einem Objekt der Klasse `factor` gespeichert sein.

```

> Nj      <- 2                                # Gruppengröße
> P       <- 2                                # Anzahl Gruppen
> Q       <- 3                                # Anzahl Messzeitpunkte
> id      <- 1:(P*Nj)                          # Blockzugehörigkeit
> DV_t1  <- round(rnorm(P*Nj, -1, 1), 2)     # AV zu t1
> DV_t2  <- round(rnorm(P*Nj,  0, 1), 2)     # AV zu t2
> DV_t3  <- round(rnorm(P*Nj,  1, 1), 2)     # AV zu t3
> IVbtw <- factor(rep(c("A", "B"), Nj))      # Gruppe: between

# Datensatz im Wide-Format
> (dfWide <- data.frame(id, IVbtw, DV_t1, DV_t2, DV_t3))

```

```

      id  IVbtw  DV_t1  DV_t2  DV_t3
1 1      A -1.64   0.01  1.31
2 2      B -0.81 -1.23  1.59
3 3      A -1.43 -0.80  0.68
4 4      B -1.79 -0.13 -0.14

# Variablen für Long-Format
> idL    <- factor(rep(id, Q))                      # Faktor ID-Code
> DVl    <- c(DV_t1, DV_t2, DV_t3)                  # gemeinsamer AV-Vektor
> IVwth <- factor(rep(1:3, each=P*Nj))              # Zeitpunkt: within
> IVbtwL <- rep(IVbtw, times=Q)                     # Gruppe: between

# Datensatz im Long-Format
> dfLong <- data.frame(id=idL, IVbtw=IVbtwL, IVwth=IVwth, DV=DVl)
> dfLong[order(dfLong$id), ]                         # sortierte Ausgabe
      id  IVbtw  IVwth     DV
1 1      A      1 -1.64
5 1      A      2  0.01
9 1      A      3  1.31
2 2      B      1 -0.81
6 2      B      2 -1.23
10 2     B      3  1.59
3 3      A      1 -1.43
7 3      A      2 -0.80
11 3     A      3  0.68
4 4      B      1 -1.79
8 4      B      2 -0.13
12 4     B      3 -0.14

```

`reshape()` bietet die Möglichkeit, einen Datensatz ohne manuelle Zwischenschritte zwischen Wide- und Long-Format zu transformieren.¹⁶

```

> reshape(data=Datensatz, varying, timevar="time",
+         idvar="id", direction=c("wide", "long"), v.names="<Name>")

```

- Zunächst wird der Datensatz unter `data` eingefügt. Um ihn vom Wide- ins Long-Format zu transformieren, muss das Argument `direction="long"` gesetzt werden.
- Daneben ist unter `varying` anzugeben, welche Variablen im Wide-Format dieselbe AV zu unterschiedlichen Messzeitpunkten repräsentieren. Die Variablen werden im Long-Format über unterschiedliche Ausprägungen der neu gebildeten Variable `time` identifiziert, deren Name über das Argument `timevar` auch selbst festgelegt werden kann. `varying` benötigt eine Liste, deren Komponenten Vektoren mit Variablennamen oder Spaltenindizes sind. Jeder Vektor gibt dabei eine Gruppe von Variablen an, die jeweils zu einer AV gehören.¹⁷

¹⁶Das Paket `reshape2` (Wickham, 2007) stellt weitere spezialisierte Möglichkeiten zur Transformation zwischen beiden Organisationsformen bereit. Beispiele hierfür finden sich bei Spector (2008).

¹⁷Im Fall zweier AVn, für die jeweils eine Gruppe von zwei Spalten im Wide-Format vorhanden ist, könnte das Argument also `varying=list(c("DV1_t1", "DV1_t2"), c("DV2_t1", "DV2_t2"))` lauten.

- Besitzt der Datensatz eine Variable, die codiert, von welcher Person ein Wert stammt, kann sie im Argument `idvar` genannt werden. Andernfalls wird eine solche Variable auf Basis der Zeilenindizes gebildet und trägt den Namen `id`. Auch andere Variablen von `data`, die pro Messzeitpunkt zwischen den Personen variieren, gelten als `idvar`, dies trifft etwa auf die Ausprägung von Zwischen-Gruppen Faktoren zu. Im Fall mehrerer solcher Variablen sind diese als Vektor von Variablennamen anzugeben.
- Der Name der Variable im Long-Format mit den Werten der AV kann über das Argument `v.names` bestimmt werden.

```
> resLong <- reshape(dfWide, varying=c("DV_t1", "DV_t2", "DV_t3"),
+                     direction="long", idvar=c("id", "IVbtw"), v.names="DV")

# erste Zeilen der sortierten Ausgabe, identisch zu dfLong
> head(resLong[order(resLong$id), ])
   id IVbtw time    DV
1. A.1 1      A    1 -1.68
1. A.2 1      A    2  0.78
1. A.3 1      A    3  2.73
2. B.1 2      B    1 -2.52
2. B.2 2      B    2  0.69
2. B.3 2      B    3  0.80
```

Die Variablen in der Rolle von `idvar` und `time` sollten Objekte der Klasse `factor` sein. Da `reshape()` die Variablen aber als numerische Vektoren generiert, müssen sie ggf. manuell umgewandelt werden mit:

- `< Datensatz>$<Variable> <- factor(<Datensatz>$<Variable>)`

Ist der Datensatz vom Long- ins Wide-Format zu transformieren, muss `direction="wide"` gesetzt werden. Für das Argument `v.names` wird jene Variable genannt, die die Werte der AV im Long-Format über alle Messzeitpunkte hinweg speichert. Diese Variable wird im Wide-Format auf mehrere Spalten aufgeteilt, die den Messzeitpunkten entsprechen. Dafür ist unter `timevar` anzugeben, welche Variable den Messzeitpunkt codiert. Unter `idvar` sind jene Variablen zu nennen, deren Werte die Daten der AV eines Objekts zuordnen bzw. pro Messzeitpunkt über die Objekte variieren, etwa weil sie die Ausprägung von Zwischen-Gruppen Faktoren darstellen.

```
> reshape(dfLong, v.names="DV", timevar="IVwth",
+          idvar=c("id", "IVbtw"), direction="wide")
   id IVbtw DV.1  DV.2  DV.3
1  1      A -1.64  0.01  1.31
2  2      B -0.81 -1.23  1.59
3  3      A -1.43 -0.80  0.68
4  4      B -1.79 -0.13 -0.14
```

Vorgehen bei mehreren Messwiederholungsfaktoren

Ist eine AV an denselben Objekten mehrfach in den kombinierten Bedingungen zweier Intra-Gruppen Faktoren erhoben worden (vgl. Abschn. 7.6), so muss `reshape()` zweimal angewandt

werden, um die Daten vom Wide- ins Long-Format zu transformieren. Im ersten Schritt werden die zu unterschiedlichen Bedingungen des ersten Faktors gehörenden Spalten zusammengefasst, im zweiten Schritt diejenigen des zweiten.

Im Beispiel seien an vier Personen in jeder Stufenkombination der UV1 mit drei und der UV2 mit zwei Messzeitpunkten Werte einer AV erhoben worden.

```
> N      <- 4                                # Anzahl Personen
> id    <- 1:N                               # ID-Code
> t_11 <- round(rnorm(N,  8,  2), 2)        # AV zu t1-1
> t_12 <- round(rnorm(N, 10,  2), 2)        # AV zu t1-2
> t_21 <- round(rnorm(N, 13,  2), 2)        # AV zu t2-1
> t_22 <- round(rnorm(N, 15,  2), 2)        # AV zu t2-2
> t_31 <- round(rnorm(N, 13,  2), 2)        # AV zu t3-1
> t_32 <- round(rnorm(N, 15,  2), 2)        # AV zu t3-2

# Datensatz im Wide-Format
> dfW <- data.frame(id, t_11, t_12, t_21, t_22, t_31, t_32)

# Transformation ins Long-Format bzgl. IV1
> (dfL1 <- reshape(dfW, direction="long", timevar="IV1",
+                   varying=list(c("t_11", "t_21", "t_31"), c("t_12", "t_22", "t_32")),
+                   idvar="id", v.names=c("IV2-1", "IV2-2")))
   IV1  IV2-1  IV2-2  id
1.1  1  11.59  10.27  1
2.1  1   7.28  10.63  2
3.1  1   8.43  9.74  3
4.1  1   5.05  12.58  4
1.2  2  11.96  15.27  1
2.2  2  14.97  11.47  2
3.2  2  11.96  11.39  3
4.2  2  19.12  15.45  4
1.3  3  14.40  16.18  1
2.3  3  13.48  12.06  2
3.3  3  17.34  11.62  3
4.3  3  12.55  14.76  4
```

Da IV1 nun wie `id` pro Messzeitpunkt von IV2 über die Personen variiert, muss die Variable im zweiten Schritt ebenfalls unter `idvar` genannt werden.

```
> dfL2 <- reshape(dfL1, varying=c("IV2-1", "IV2-2"),
+                   direction="long", timevar="IV2",
+                   idvar=c("id", "IV1"), v.names="DV")

> head(dfL2)                                     # erste Einträge von dfL2
   id  IV1  IV2      DV
1.1.1 1     1     1  9.26
2.1.1 2     1     1  8.93
```

```
3.1.1 3     1     1    8.12
4.1.1 4     1     1    3.57
1.2.1 1     2     1   13.78
2.2.1 2     2     1   15.85
```

Auch die umgekehrte Transformation vom Long- ins Wide-Format benötigt zwei Schritte, wenn zwei Intra-Gruppen Faktoren vorhanden sind.

```
# Schritt 1: Stufen der IV2 in Spalten aufteilen
> dfW1 <- reshape(dfL2, v.names="DV", timevar="IV2",
+                   idvar=c("id", "IV1"), direction="wide")

# Schritt 2: Stufen der IV1 in Spalten aufteilen
> dfW2 <- reshape(dfW1, v.names=c("DV.1", "DV.2"),
+                   timevar="IV1", idvar="id", direction="wide")

# Vergleich mit ursprünglichem Datensatz im Wide-Format
> all.equal(dfW, dfW2, check.attributes=FALSE)
[1] TRUE
```

3.4 Daten aggregieren

Die folgenden Abschnitte stellen dar, wie allgemein Funktionen auf Variablen aus Datensätzen angewendet werden können. Häufig sind zudem Daten nach Gruppen aufzuteilen, für diese dann Kennwerte zu berechnen und letztere wieder zusammenzuführen.¹⁸

3.4.1 Funktionen auf Variablen anwenden

`lapply()` (*list apply*) verallgemeinert die Funktionsweise von `apply()` auf Listen und Datensätze, um eine Funktion auf deren Komponenten, also Variablen anzuwenden. Hier entfällt die Angabe, ob die Funktion auf Zeilen oder Spalten angewendet werden soll – es sind immer die Komponenten der Liste, entsprechend die Variablen bzw. Spalten eines Datensatzes.

```
> lapply(X=<Liste>, FUN=<Funktion>, ...)
```

Das Ergebnis von `lapply()` ist eine Liste mit ebenso vielen Komponenten wie sie die Liste X enthält. Ist FUN nur sinnvoll auf numerische Variablen anwendbar, können diese etwa zunächst mit `subset(..., select=<Indizes>)` extrahiert werden.

```
# Mittelwerte der numerischen Variablen
> numDf    <- subset(myDf1, select=c(age, IQ, rating))
> (myList <- lapply(numDf, mean))
$age
[1] 26.41667
```

¹⁸Das Paket `dplyr` enthält spezialisierte Funktionen, die diese Arbeitsschritte systematisieren und besonders bequem durchführbar machen.

```
$IQ
[1] 97.25
```

```
$rating
[1] 2.583333
```

`sapply()` (*simplified apply*) arbeitet wie `lapply()`, gibt aber nach Möglichkeit keine Liste, sondern einen einfacher zu verarbeitenden Vektor mit benannten Elementen aus. Gibt FUN pro Aufruf mehr als einen Wert zurück, ist das Ergebnis eine Matrix, deren Spalten aus diesen Werten gebildet sind.

```
# range der numerischen Variablen
> sapply(numDf, range)
      age   IQ rating
[1,] 20   82     0
[2,] 35  122     5
```

Durch die Ausgabe eines Vektors eignet sich `sapply()` z. B. dazu, aus einem Datensatz jene Variablen zu extrahieren, die eine bestimmte Bedingung erfüllen – etwa einen numerischen Datentyp besitzen. Der Ergebnisvektor kann später zur Indizierung der Spalten Verwendung finden.¹⁹

```
> (numIdx <- sapply(myDf1, is.numeric))           # numerische Variable?
  id   sex group age   IQ rating
TRUE FALSE FALSE TRUE  TRUE    TRUE

> dataNum <- subset(myDf1, select=numIdx)         # nur numerische Variablen
> head(dataNum)
  id age   IQ rating
1  1 26 112     1
2  2 30 122     3
3  3 25  95     5
4  4 34 102     5
5  5 22  82     2
6  6 24 113     0
```

Eine vereinfachte Form von `sapply()` ist als `replicate()` Funktion verfügbar, die lediglich dafür sorgt, dass derselbe Ausdruck expr mehrfach (n mal) wiederholt wird. Die Ausgabe erfolgt als Matrix mit n Spalten und so vielen Zeilen, wie expr pro Ausführung Werte erzeugt.

```
> replicate(n=<Anzahl>, expr=<Befehl>)

> replicate(6, round(rnorm(4), 2))
 [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
```

¹⁹ `sapply()` ist auch für jene Fälle nützlich, in denen auf jedes Element eines Vektors eine Funktion angewendet werden soll, diese Funktion aber nicht vektorisiert ist – d. h. als Argument nur einen einzelnen Wert, nicht aber Vektoren akzeptiert. In diesem Fall betrachtet `sapply()` jedes Element des Vektors als eigene Variable, die nur einen Wert beinhaltet.

```
[1,] -0.31 -0.17  0.35 -0.46 -1.23 -2.39
[2,]  0.73 -1.12  0.60 -0.61 -0.07  0.58
[3,] -0.16  0.64  0.46 -1.21  0.20 -0.81
[4,]  0.87  0.49  0.04 -1.38  0.05  1.83
```

Mit `do.call()` ist eine etwas andere automatisierte Anwendung einer Funktion auf die Komponenten einer Liste bzw. auf die Variablen eines Datensatzes möglich. Während `lapply()` eine Funktion so häufig aufruft, wie Variablen vorhanden sind und dabei jeweils eine Variable als Argument übergibt, geschieht dies bei `do.call()` nur einmal, dafür aber mit mehreren Argumenten.

```
> do.call(what="Funktionsname", args=Liste)
```

Unter `what` ist die aufzurufende Funktion zu nennen, unter `args` deren Argumente in Form einer Liste, wobei jede Komponente von `args` ein Funktionsargument liefert. Ist von vornherein bekannt, welche und wie viele Argumente `what` erhalten soll, könnte `do.call()` auch durch einen einfachen Aufruf von `Funktion(Liste[1], Liste[2], ...)` ersetzt werden, nachdem die Argumente als Liste zusammengestellt wurden. Der Vorteil der Konstruktion eines Funktionsaufrufs aus dem Funktionsnamen einerseits und den Argumenten andererseits tritt jedoch dann zutage, wenn sich Art oder Anzahl der Argumente erst zur Laufzeit der Befehle herausstellen – etwa weil die Liste selbst erst mit vorangehenden Befehlen dynamisch erzeugt wurde.

Aus einer von `lapply()` zurückgegebenen Liste ließe sich damit wie folgt ein Vektor machen, wie ihn auch `sapply()` zurückgibt:

```
# äquivalent zu
# c(id=myList[[1]], age=myList[[2]], IQ=myList[[3]], rating=myList[[4]])
> do.call("c", myList)
      id      age      IQ      rating
6.500000 26.416667 97.250000 2.583333
```

Sind die Komponenten von `args` benannt, behalten sie ihren Namen bei der Verwendung als Argument für die unter `what` genannte Funktion bei. Damit lassen sich beliebige Funktionsaufrufe samt zu übergebender Daten und weiterer Optionen konstruieren: Alle späteren Argumente werden dafür als Komponenten in einer Liste gespeichert, wobei die Komponenten die Namen erhalten, die die Argumente der Funktion `what` tragen.

```
> work <- factor(sample(c("home", "office"), 20, replace=TRUE))
> hiLo <- factor(sample(c("hi", "lo"), 20, replace=TRUE))
> group <- factor(sample(c("A", "B"), 20, replace=TRUE))
> tab <- table(work, hiLo, group)      # 3D-Kreuztabelle der Faktoren

# wandle 3D-Kreuztabelle mit ftable um -> lege fest, welche Faktoren in
# Zeilen (row.vars), welche in Spalten (col.vars) stehen sollen
> argLst <- list(tab, row.vars="work", col.vars=c("hiLo", "group"))
> do.call("ftable", argLst)
    hiLo   hi     lo
    group A B     A B
```

```
work
home      1  3   1  3
office    2  3   2  5
```

3.4.2 Funktionen für mehrere Variablen anwenden

`lapply()` und `sapply()` wenden Funktionen auf Variablen eines Datensatzes an, die nur die Daten jeweils einer Variable als Basis für Berechnungen nutzen. `mapply()` verallgemeinert dieses Prinzip auf Funktionen, die aus mehr als einer einzelnen Variable Kennwerte berechnen. Dies ist insbesondere für viele inferenzstatistische Tests der Fall, die etwa in zwei Variablen vorliegende Daten aus zwei Stichproben hinsichtlich verschiedener Kriterien vergleichen.

```
> mapply(FUN=<Funktion>, <Datensatz1>, <Datensatz2>, ...,
+         MoreArgs=<Liste mit Optionen für FUN>)
```

Die anzuwendende Funktion ist als erstes Argument `FUN` zu nennen. Es folgen so viele Datensätze, wie `FUN` Eingangsgrößen benötigt. Im Beispiel einer Funktion für zwei Variablen verrechnet die Funktion schrittweise zunächst die erste Variable des ersten zusammen mit der ersten Variable des zweiten Datensatzes, dann die zweite Variable des ersten zusammen mit der zweiten Variable des zweiten Datensatzes, etc. Sollen an `FUN` weitere Argumente übergeben werden, kann dies mit dem Argument `MoreArgs` in Form einer Liste geschehen.

Im Beispiel soll ein *t*-Test für zwei unabhängige Stichproben für jeweils alle Variablen-Paare zweier Datensätze berechnet werden (vgl. Abschn. 7.2). Dabei soll im *t*-Test eine gerichtete Hypothese getestet (`alternative="less"`) und von Varianzhomogenität ausgegangen werden (`var.equal=TRUE`). Die Ausgabe wird hier verkürzt dargestellt.

```
> N      <- 100
> x1    <- rnorm(N, 10, 10)           # Variablen für ersten Datensatz
> y1    <- rnorm(N, 10, 10)
> x2    <- x1 + rnorm(N, 5, 4)        # Variablen für zweiten Datensatz
> y2    <- y1 + rnorm(N, 10, 4)
> myDf2 <- data.frame(x1, y1)
> myDf3 <- data.frame(x2, y2)
> mapply(t.test, myDf2, myDf3, MoreArgs=list(alternative="less",
+                                               var.equal=TRUE))
      x1
statistic -1.925841
parameter 198
p.value 0.02777827
alternative "less"
method     "Two Sample t-test"          # ...
      y1
statistic -33.75330
parameter 198
p.value 2.291449e-84
```

```
alternative "less"
method      "Two Sample t-test"                      # ...
```

3.4.3 Funktionen getrennt nach Gruppen anwenden

Nachdem Datensätze mit `split()` geteilt wurden, liegen die Daten aus den Bedingungen separat vor und können deshalb getrennt verarbeitet werden, etwa zur Berechnung von Kennwerten pro Gruppe. Hierfür eignen sich die Funktionen `lapply()` und `sapply()`, die dieselbe Operation für jede Komponente einer Liste durchführen. Sind nur getrennt nach Gruppen zu berechnende Kennwerte einzelner Variablen von Interesse, ist jedoch `tapply()` das einfachere Mittel.

```
# berechne Mittelwerte von IQ getrennt nach Gruppen
> (splitRes <- split(myDf1, myDf1$group))                  # ...
> sapply(splitRes, function(x) mean(x$IQ))
    CG      T      WL
105.50 101.00  85.25

> tapply(myDf1$IQ, myDf1$group, mean)                         # Kontrolle ...
```

Um für Variablen eines Datensatzes Kennwerte nicht nur über alle Beobachtungen hinweg, sondern getrennt nach Gruppen zu berechnen, sind auch `aggregate()` und `by()` vorhanden.

```
> aggregate(x=<Datensatz>, by=<Liste mit Faktoren>, FUN=<Funktion>, ...)
> by(data=<Datensatz>, INDICES=<Liste mit Faktoren>, FUN=<Funktion>, ...)
```

Dabei wird ein Kennwert für die Variablen des unter `x` bzw. `data` anzugebenden Datensatzes berechnet. Die Argumente `by` bzw. `INDICES` kontrollieren, in welche Gruppen die Beobachtungen dabei eingeteilt werden. Anzugeben ist eine Liste, die als Komponenten Gruppierungsfaktoren der Länge `nrow(x)` bzw. `nrow(data)` enthält. Mit dem Argument `FUN` wird die Funktion spezifiziert, die auf die gebildeten Gruppen in jeder Variable angewendet werden soll. Die in `by()` für `FUN` verwendete Funktion muss einen Datensatz als Argument akzeptieren, während `FUN` in `aggregate()` intern mit einem einzelnen Vektor aufgerufen wird. `aggregate()` und `by()` ähneln der Funktion `tapply()`, unterscheiden sich jedoch von ihr durch die Gruppenbildung und anschließende Funktionsanwendung auf mehrere Variablen gleichzeitig.

`FUN` wird auf alle Variablen des übergebenen Datensatzes pro Gruppe angewendet, auch wenn für einzelne von ihnen, etwa Faktoren, die Berechnung numerischer Kennwerte nicht möglich oder nicht sinnvoll ist. Um dies von vornherein zu verhindern, ist der Datensatz auf eine geeignete Teilmenge seiner Variablen zu beschränken.

```
# pro Bedingungskombination: Mittelwert für Alter, IQ und Rating
> numDf <- subset(myDf1, select=c("age", "IQ", "rating"))
> aggregate(numDf, list(myDf1$sex, myDf1$group), mean)
  Group.1 Group.2   age     IQ rating
1       m      CG 25.00 103.00     3
2       f      CG 24.00 113.00     0
3       m        T 31.00  97.00     4
4       f        T 23.50 105.00     1
```

5	m	WL	27.25	85.25	3
---	---	----	-------	-------	---

Als Alternative zu den Optionen `x` und `by` erlaubt es `aggregate()` auch, die zu analysierenden Variablen und Gruppierungsfaktoren als (ggf. multivariat formulierte) Modellformel zu definieren (vgl. Abschn. 5.2). Die Variablen, für die der mit `FUN` bezeichnete Kennwert zu berechnen ist, stehen dabei links von der Tilde `~`, die Gruppierungsfaktoren durch ein `+` verbunden rechts von ihr. Die Variablen können mit ihrem einfachen Namen bezeichnet werden, sofern der sie enthaltende Datensatz für das Argument `data` genannt ist. Den Kennwert für alle Daten erhält man, wenn rechts der Tilde nur `~ 1` steht.

```
# äquivalent ...
> aggregate(cbind(age, IQ, rating) ~ sex + group, FUN=mean, data=myDf1)

# Gesamtmittelwert jeweils von age, IQ und rating
> aggregate(cbind(age, IQ, rating) ~ 1, FUN=mean, data=myDf1)
      age     IQ   rating
1 26.41667 97.25 2.583333
```

Während das Ergebnis von `aggregate()` ein Datensatz ist, erfolgt die Ausgabe von `by()` als Objekt der Klasse `by`, das im Fall eines einzelnen Gruppierungsfaktors eine Liste, sonst ein array ist. Da `mean()` keinen Datensatz als Argument akzeptiert, wird für `FUN` hier eine selbst definierte Funktion verwendet (vgl. 15.2).

```
# pro Bedingungskombination: Mittelwert für Alter, IQ und Rating
> by(numDf, list(myDf1$sex, myDf1$group), FUN=function(x) sapply(x, mean))
: f
: CG
  age     IQ   rating
  24    113     0

-----
: m
: CG
  age     IQ   rating
  25    103     3
# ...                                # gekürzte Ausgabe
```

Kapitel 4

Befehle und Daten verwalten

Für Datenanalysen, die über wenige Teilschritte hinausgehen, ist die interaktive Arbeitsweise meist nicht sinnvoll, in der sich eingegebene Befehle mit der von R erzeugten Ausgabe auf der Konsole abwechseln. Stattdessen lässt sich die Auswertung automatisieren, indem alle Befehle zunächst zeilenweise in eine als *Skript* bezeichnete Textdatei geschrieben werden, die dann ihrerseits von R komplett oder in Teilen ausgeführt wird. Analoges gilt für die Verwaltung empirischer Daten: Gewöhnlich werden diese nicht von Hand auf der Konsole eingegeben, sondern in separaten Dateien gespeichert – sei es in R, in Programmen zur Tabellenkalkulation oder in anderen Statistikpaketen. Vergleiche Abschn. 4.3 für die Form der Pfadangaben zu Dateien in den folgenden Abschnitten.

4.1 Befehlssequenzen im Editor bearbeiten

Um Befehlsskripte zu erstellen, bietet RStudio einen Texteditor, der sich mit File: New File: R Script öffnen lässt und daraufhin bereit für die Eingabe von Befehlszeilen ist. Der Texteditor erstellt automatisch zu jeder öffnenden Klammer eine passende schließende und hebt R-Befehle farblich hervor (*syntax-highlighting*). Mit der Tastenkombination **Strg+R** wird der Befehl in derjenigen Zeile von R ausgeführt, in der sich der Cursor gerade befindet (icon Run). Um in einem Schritt mehrere Befehlszeilen auswerten zu lassen, markiert man diese und führt sie ebenfalls mit **Strg+R** aus. Um das komplette Skript in Gänze ausführen zu lassen, kann das icon Source verwendet werden. Verursacht einer der auszuführenden Befehle eine Fehlermeldung, unterbricht dies die Verarbeitung, ggf. folgende Befehle werden dann also nicht ausgewertet. Einfache Warnungen werden dagegen gesammelt am Schluss aller Ausgaben genannt. Befehlsskripte lassen sich über File: Save as... speichern und über File: Open File... laden.

In externen Dateien gespeicherte Skripte lassen sich in der Konsole mit `source("<Dateiname>")` einlesen, wobei R die enthaltenen Befehle ausführt.¹ Befindet sich die Skriptdatei nicht im aktiven Arbeitsverzeichnis, muss der vollständige Pfad zum Skript zusätzlich zu seinem Namen mit angegeben werden, z. B. `source("c:/work/r/skript.r")`.² Wird dabei das Argument `echo=TRUE` gesetzt, werden die im Skript enthaltenen Befehle selbst mit auf der Konsole ausgegeben, andernfalls erscheint nur die Ausgabe dieser Befehle.

Es bieten sich u. a. folgende Vorteile, mit in Textdateien gespeicherten Skripten zu arbeiten:

¹ Es ist nicht notwendig, R im interaktiven Modus zu starten, um ein Befehlsskript ausführen zu lassen, dafür existiert auch ein *Batch-Modus*, vgl. `?Rscript`.

² Wird das einzlesende Skript nicht gefunden, ist zunächst mit `dir()` zu prüfen, ob das von R durchsuchte Verzeichnis (ohne explizite Angabe eines Pfades ist es das mit `getwd()` angezeigte Arbeitsverzeichnis) auch tatsächlich das Skript enthält.

- Der Überblick über alle auszuführenden Befehle wird erleichtert, zudem können die Auswertungsschritte gedanklich nachvollzogen werden.
- Komplexe Auswertungen lassen sich in kleinere Teilschritte zerlegen. Diese können einzeln nacheinander oder in Teilsequenzen ausgeführt werden, um Zwischenergebnisse auf ihre Richtigkeit zu prüfen.
- Man kann ein einmal erstelltes Skript immer wieder ausführen lassen. Dies ist insbesondere bei der Fehlersuche und bei nachträglichen Veränderungswünschen, etwa an Grafiken, hilfreich: Anders als z. B. in Programmen zur Tabellenkalkulation müssen so im Fall von anderen Überschriften oder Achsenkalierungen nicht viele schon bestehende Grafiken mit immer denselben Schritten einzeln geändert werden. Stattdessen reicht es aus, das Skript einmal zu ändern und neu auszuführen, um die angepassten Grafiken zu erhalten.
- Ein für die Auswertung eines Datensatzes erstelltes Skript lässt sich häufig mit nur geringen Änderungen für die Analyse anderer Datensätze anpassen. Diese Wiederverwendbarkeit einmal geleisteter Arbeit ist bei rein grafisch zu bedienenden Programmen nicht gegeben und spart auf längere Sicht Zeit. Zudem vermeidet eine solche Vorgehensweise Fehler, wenn geprüfte und bewährte Befehlssequenzen kopiert werden können.
- Ein erstelltes Skript dokumentiert gleichzeitig die Auswertungsschritte und garantiert damit die Reproduzierbarkeit der gewonnenen Ergebnisse statistischer Analysen.
- Skripte lassen sich zusammen mit dem Datensatz, für dessen Auswertung sie gedacht sind, an Dritte weitergeben. Neben dem Aspekt der so möglichen Arbeitsteilung kann der Auswertungsvorgang von anderen Personen auf diese Weise nachvollzogen und kontrolliert werden. Dies ist i. S. einer größeren Auswertungsobjektivität sinnvoll.³
- Zudem lassen sich von R nicht als Befehl interpretierte Kommentare einfügen, z. B. um die Bedeutung der Befehlssequenzen zu erläutern. Kommentare sind dabei alle Zeilen bzw. Teile von Zeilen, die mit einem # Symbol beginnen. Ihre Verwendung ist empfehlenswert, damit auch andere Personen schnell erfassen können, was Befehle bedeuten oder bewirken sollen. Aber auch für den Autor des Skripts selbst sind Kommentare hilfreich, wenn es längere Zeit nach Erstellen geprüft oder für eine andere Situation angepasst werden soll.

4.2 Daten importieren und exportieren

Empirische Daten können auf verschiedenen Wegen in R verfügbar gemacht werden. Zunächst ist es möglich, Werte durch Zuweisungen etwa in Vektoren zu speichern und diese dann zu Datensätzen zusammenzufügen. Bequemer und übersichtlicher ist die Benutzung des in R integrierten Dateneditors (vgl. Abschn. 4.2.1). Häufig liegen Datensätze aber auch in Form von mit anderen Programmen erstellten Dateien vor (vgl. Abschn. 4.2.4). R bietet die Möglichkeit, auf verschiedenste Datenformate zuzugreifen und in diesen auch wieder Daten abzulegen. Immer sollte dabei überprüft werden, ob die Daten auch tatsächlich korrekt transferiert wurden. Zudem empfiehlt es sich, nie mit den Originaldaten selbst zu arbeiten. Stattdessen sollten immer nur Kopien eines Referenz-Datensatzes verwendet werden, um diesen gegen unbeabsichtigte

³Da R mit `shell()` auch auf Funktionen des Betriebssystems zugreifen kann, sollten aus Sicherheitsgründen nur geprüfte Skripte aus vertrauenswürdiger Quelle ausgeführt werden.

Veränderungen zu schützen. Dateneingabe sowie der Datenaustausch mit anderen Programmen werden vertieft im Manual „R Data Import/Export“ (R Development Core Team, 2014b) sowie von Muenchen (2011) und Spector (2008) behandelt.

4.2.1 Daten im Editor eingeben

Bereits im workspace vorhandene Datensätze oder einzelne Variablen können unter Windows und MacOS über den in R integrierten Dateneditor geändert werden, der mit `edit(<Objekt>)` aufgerufen wird. Innerhalb des Editors können Zellen mit der Maus ausgewählt und dann durch entsprechende Tastatureingaben mit Werten gefüllt werden – eine leere Zelle steht dabei für einen fehlenden Wert. Ebenso lassen sich durch einen Klick auf die Spaltenköpfe Name und Datentyp der Variablen neu festlegen. Während der Dateneditor geöffnet ist, bleibt die Konsole für Eingaben blockiert. Beim Schließen des Dateneditors liefert `edit()` als Rückgabewert das Objekt mit allen ggf. geänderten Werten. Wichtig ist, dieses zurückgelieferte Objekt dem zu ändernden Objekt beim Aufruf zuzuweisen, damit die Änderungen auch gespeichert werden. Wurde dies vergessen, ist der geänderte Datensatz noch als `.Last.value` vorhanden, solange kein neuer output erzeugt wird.

Der Befehl `fix(<Objekt>)` ähnelt `edit()`, fasst das Speichern des bearbeiteten Objekts aber bereits mit ein und ist daher meist zu bevorzugen. Aus diesem Grund lassen sich mit `fix()` auch neuen Datensätze erstellen, wie dies mit `edit(data.frame())` über den Umweg eines verschachtelt im Aufruf erzeugten leeren Datensatzes möglich ist.

```
> myDf <- data.frame(IV=factor(rep(c("A", "B"), 5)), DV=rnorm(10))
> myDf <- edit(myDf)           # Zuweisung des Ergebnisses erforderlich
> fix(myDf)                  # Zuweisung nicht erforderlich
> newDf <- edit(data.frame()) # editiere leeren Datensatz
```

4.2.2 Datentabellen im Textformat

Für jeden Import in R sollten Daten so organisiert sein, dass sich die Variablen in den Spalten und die Werte jeweils eines Beobachtungsobjekts in den Zeilen befinden. Für alle Variablen sollten gleich viele Beobachtungen vorliegen, so dass sich insgesamt eine rechteckige Datenmatrix ergibt. Bei fehlenden Werten ist es am günstigsten, sie konsistent mit einem expliziten Code zu kennzeichnen, der selbst kein möglicher Wert ist. Weiter ist darauf zu achten, dass Variablennamen den R-Konventionen entsprechen und beispielsweise kein #, %, ' oder Leerzeichen enthalten (vgl. Abschn. 1.3.1).

Mit `read.table()` werden in Textform vorliegende Daten geladen und in einem Objekt der Klasse `data.frame` ausgegeben. Wichtige Argumente von `read.table()` sind in Tab. 4.1 dargestellt. RStudio vereinfacht den Import von Textdaten über das Menü Tools: Import Dataset.

Tabelle 4.1: Wichtige Argumente von `read.table()`

Argument	Bedeutung
----------	-----------

Tabelle 4.1: (Forts.)

<code>file</code>	(ggf. Pfad und) Name der einzulesenden Quelle bzw. des zu schreibenden Ziels (meist eine Datei), in Anführungszeichen gesetzt ⁴
<code>header</code>	Wenn in der einzulesenden Quelle Spaltennamen vorhanden sind, muss <code>header=TRUE</code> gesetzt werden (Voreinstellung ist <code>FALSE</code>)
<code>sep</code>	Trennzeichen zwischen zwei Spalten in <code>file</code> . Voreinstellung ist jeglicher zusammenhängender whitespace (Leerzeichen oder Tabulatoren), unabhängig davon, wie viele davon aufeinander folgen. Andere häufig verwendete Werte sind das Komma (",") oder das Tabulatorzeichen ("\t") ⁵
<code>strip.white</code>	Bestimmt, ob whitespace vor bzw. nach einem Wert entfernt werden soll. In der Voreinstellung <code>FALSE</code> geschieht dies bei numerischen Werten, nicht aber bei Zeichenketten. Sollte auf <code>TRUE</code> gesetzt werden, wenn auch bei <code>sep</code> von der Voreinstellung abgewichen wird
<code>dec</code>	Das in der Datei verwendete Dezimaltrennzeichen, Voreinstellung ist der Punkt (<code>dec=".."</code>)
<code>colClasses</code>	Vektor, der für jede Spalte der einzulesenden Quelle den Datentyp angibt, z. B. <code>c("numeric", "logical", ...)</code> . Mit <code>NULL</code> können Spalten auch übersprungen, also vom Import ausgeschlossen werden, es müssen aber für alle Spalten Angaben vorhanden sein. In der Voreinstellung <code>NA</code> bestimmt R selbst die Datentypen, was bei großen Datenmengen langsamer ist
<code>na.strings</code>	Vektor mit den zur Codierung fehlender Werte verwendeten Zeichenketten. Voreinstellung ist " <code>NA</code> "
<code>stringsAsFactors</code>	Variablen mit Zeichenketten als Werten werden automatisch in Gruppierungsfaktoren (<code>factor</code>) konvertiert (Voreinstellung <code>TRUE</code>). Sollen solche Variablen als <code>character</code> Vektoren gespeichert werden, ist das Argument auf <code>FALSE</code> zu setzen (vgl. Abschn. 3.2.1)

Für das Argument `file` können nicht nur lokal gespeicherte Dateien angegeben werden: Die Funktion liest unter Windows mit `file="clipboard"` auch Werte aus der Zwischenablage, die dort in einem anderen Programm etwa mit Strg+C hineinkopiert wurden.⁶ Ebenso liest sie wie

⁴Werden die einzulesenden Daten von R nicht gefunden, ist zunächst mit `dir()` zu prüfen, ob das von R durchsuchte Verzeichnis (ohne explizite Angabe eines Pfades ist es das mit `getwd()` angezeigte Arbeitsverzeichnis) auch jenes ist, das die Datei enthält.

⁵Sobald für das Argument `sep` ein selbst gewählter Wert wie "\t" vergeben wird, ändert sich die Bedeutung dieses Zeichens: Tauchen in der Datei dann etwa zwei Tabulatoren hintereinander auf, interpretiert R dies als eine leere Zelle der Datenmatrix, also als fehlenden Wert. Ebenso gelten zwei nur durch ein Leerzeichen getrennte Werte nicht mehr als zwei Zellen.

⁶Unter MacOS ist `file=pipe("pbpaste")` zu verwenden.

`scan()` Daten von der Konsole, wenn `file=stdin()` verwendet wird.

```
> (xDf <- read.table(file=stdin(), header=TRUE))
0: id group rating
1: 1 A 3
2: 2 A 1
3: 3 B 5
4:
  id group rating
1 1     A      3
2 2     A      1
3 3     B      5
```

Online verfügbare Dateien können mit `file=url("<URL>")` direkt von einem Webserver geladen werden.⁷ Anders als in Webbrowsern muss dabei der Protokollteil der Adresse (etwa `http://` oder `ftp://`) explizit genannt werden, also z. B.

- `file=url("http://www.server.de/datei.txt")`

Der Abschnitt **Web Technologies and Services** der CRAN Task Views (Chamberlain, Ram, Gandrud & Mair, 2014) stellt Pakete und Funktionen vor, um Daten direkt aus Webseiten zu extrahieren.

Zum Speichern von Objekten in Textdateien dient `write.table()`.

```
> write.table(x=<Objekt>, file=<Dateiname>, sep=" ", dec=".",
+               row.names=TRUE, col.names=TRUE, quote=TRUE)
```

Die Funktion akzeptiert als Argumente u. a. `file`, `sep` und `dec` mit derselben Bedeutung wie bei `read.table()`. Statt in eine Datei kann `write.table()` mit `file="clipboard"` eine begrenzte Menge von Daten auch in die Zwischenablage schreiben, woraufhin sie in anderen Programmen mit Strg+V eingefügt werden können. Über die Argumente `row.names` und `col.names` wird festgelegt, ob Zeilen- und Spaltennamen mit in die Datei geschrieben werden sollen (Voreinstellung für beide ist `TRUE`). Zeichenketten werden in der Ausgabe in Anführungszeichen gesetzt, sofern nicht das Argument `quote=FALSE` gesetzt wird.

Wenn z. B. der Datensatz `myDf` im aktuellen Arbeitsverzeichnis in Textform gespeichert und später wieder eingelesen werden soll, so lauten die Befehle:

```
> write.table(myDf, file="data.txt")
> myDf <- read.table("data.txt", header=TRUE)
> str(myDf)
'data.frame': 10 obs. of 2 variables:
$ IV: Factor w/ 2 levels "A","B": 1 2 1 2 1 2 1 2 1 2
$ DV: num 0.425 -1.224 -0.572 -0.738 -1.753 ...
```

Das von `read.table()` ausgegebene Objekt besitzt die Klasse `data.frame`, selbst wenn mit `write.table()` eine Matrix gespeichert wurde. Ist dies unerwünscht, muss der Datensatz explizit z. B. mit `as.matrix(<Objekt>)` in eine andere Klasse konvertiert werden.

⁷Statt einer Datei akzeptieren die meisten Funktionen für das Argument `file` allgemein eine `connection`, bei der es sich etwa auch um die Verbindung zu einem Vektor von Zeichenketten handeln kann, vgl. `?textConnection`.

Für Daten in Textform aus sehr großen Dateien arbeitet `fread()` aus dem Paket `data.table` deutlich schneller als `read.table()`.

4.2.3 R-Objekte

Eine andere Möglichkeit zur Verwaltung von Objekten in externen Dateien bieten `save(<Daten>, file=<Dateiname>)` zum Speichern und `load(file=<Dateiname>)` zum Öffnen. Unter `<Daten>` können dabei verschiedene Objekte durch Komma getrennt angegeben werden. Alternativ lässt sich das Argument `list` verwenden, das einen Vektor mit den Namen der zu speichernden Objekte akzeptiert. `list=ls()` würde also alle sichtbaren Objekte des workspace übergeben.

Die Daten werden in einem R-spezifischen, aber plattformunabhängigen Format gespeichert, bei dem Namen und Klassen der gespeicherten Objekte erhalten bleiben. Deshalb ist es nicht notwendig, das Ergebnis von `load()` einem Objekt zuzuweisen; die gespeicherten Objekte werden unter ihrem Namen wiederhergestellt. `save.image(file=".RData")` speichert alle Objekte des aktuellen workspace.

```
> save(myDf, file="data.RData")      # speichere myDf im Arbeitsverzeichnis
> load("data.RData")                # lies myDf wieder ein
```

Ähnlich wie `save()` Objekte in einem binären Format speichert, schreibt `dump("Objekt", file=<Dateiname>)` die Inhalte von Objekten in eine Textdatei mit R-Befehlen, die sich auch durch gewöhnliche Texteditoren bearbeiten lässt. Auf diese Weise erzeugte Dateien lassen sich mit `source(file=<Dateiname>)` einlesen.

```
> dump("myDf", file="dumpMyDf.txt")
> source("dumpMyDf.txt")
```

4.2.4 Daten mit anderen Programmen austauschen

Wenn Daten mit anderen Programmen ausgetauscht werden sollen – etwa weil die Daten nicht in R eingegeben wurden, so ist der Datentransfer oft in Form von reinen Textdateien möglich. Diese Methode ist auch recht sicher, da sich die Daten jederzeit mit einem Texteditor inspizieren lassen und der korrekte Transfer in allen Stufen kontrolliert werden kann. In diesem Fall kommen in R meist `read.table()` und `write.table()` zum Einsatz.

Beim Import- und Export von Daten in Dateiformaten kommerzieller Programme besteht dagegen oft die Schwierigkeit, dass die Formate nicht öffentlich dokumentiert und auch versionsabhängigen Änderungen unterworfen sind. Wie genau Daten aus diesen Formaten gelesen und geschrieben werden können, ist deshalb mitunter für die Entwickler der entsprechenden R-Funktionen nicht mit Sicherheit zu ermitteln. Beim Austausch von Daten über proprietäre Formate ist aus diesem Grund Vorsicht geboten – bevorzugt sollten einfach strukturierte Datensätze verwendet werden.⁸

⁸Für den Transfer zwischen vielen hier nicht erwähnten Programmen existieren Zusatzpakete, die sich auf CRAN finden lassen (vgl. Abschn. 1.2.7). Das Programm Stat/Transfer (Circle Systems, 2014) ist eine kommerzielle Lösung, um Daten zuverlässig zwischen R, Tabellenkalkulationen, SAS, Stata und einigen anderen Formaten auszutauschen.

Programme zur Tabellenkalkulation

Wurde ein Programm zur Tabellenkalkulation (etwa Microsoft Excel oder OpenOffice Calc) zur Dateneingabe benutzt, so ist der Datentransfer am einfachsten, wenn die Daten von dort in eine Textdatei exportiert werden, wobei als Spalten-Trennzeichen der Tabulator verwendet wird. Der Transfer von Datumsangaben ist dabei jedoch fehlerträchtig. Dezimaltrennzeichen ist in Programmen zur Tabellenkalkulation für Deutschland das Komma.⁹ Um eine mit diesen Einstellungen exportierte Datei mit Spaltennamen in der ersten Zeile in R zu laden, wäre ein geeigneter Aufruf von `read.table()`:

```
> myDf <- read.table(file="Dateiname", header=TRUE, sep="\t", dec=",")
```

Programme zur Tabellenkalkulation verwenden in der Voreinstellung meist den Tabulator als Spaltentrennzeichen, ein Austausch kleinerer Datenmengen ohne Umweg über eine externe Datei ist unter Windows also auch wie folgt möglich: Zunächst wird in der Tabellenkalkulation der gewünschte Datenbereich inkl. der Variablennamen in der ersten Zeile markiert und mit Strg+C in die Zwischenablage kopiert. In R können die Daten dann so eingelesen werden:

```
> myDf <- read.table(file="clipboard", header=TRUE, sep="\t", dec=",")
```

Um einen Datensatz aus R heraus wieder einem anderen Programm verfügbar zu machen, wird er in demselben Format gespeichert – entweder in einer Datei oder bei sehr kleinen Datensätzen in der Zwischenablage. Im anderen Programm können die Daten aus der Zwischenablage dann mit Strg+V eingefügt werden.

```
> write.table(x=Datensatz, file="clipboard", sep="\t", dec=",",
+               row.names=FALSE)
```

Für Excel stellt REExcel ein *add-in* zur Verfügung, das dafür sorgt, dass R-Funktionen direkt aus Excel heraus benutzbar sind (Heiberger & Neuwirth, 2009; Neuwirth, 2014). Zudem ermöglicht es einen Datenaustausch ohne den Umweg eines Exports ins Textformat, indem es in R Funktionen zum Lesen und Schreiben von *Dateiname.xlsx* Dateien bereitstellt. Um Excel-Dateien in R zu verwenden, eignet sich auch das auf Java basierende Paket **XLConnect** (Mirai Solutions GmbH, 2014). Generell empfiehlt es sich, aus Excel zu importierende Daten dort so anzurufen, dass sie in der ersten Zeile und Spalte des ersten Tabellenblattes beginnen, wobei die Spaltennamen in der ersten Zeile stehen.

SPSS, Stata und SAS

SPSS verfügt mit den „Essentials for R“ über eine nachträglich installierbare Erweiterung, mit der R-Befehle direkt in SPSS ausgewertet werden können. Auf diese Weise lassen sich dort nicht nur in R verwaltete Datensätze nutzen, sondern auch ganze Auswertungsschritte bis hin zur Erstellung von Diagrammen in R-Syntax durchführen. Genauso erlaubt es die Erweiterung, mit SPSS erstellte Datensätze im R-Format zu exportieren. Einige der im folgenden beschriebenen Einschränkungen, die andere Methoden des Datenaustauschs mit sich bringen, treffen auf die

⁹Sofern dies nicht in den Ländereinstellungen der Windows Systemsteuerung geändert wurde.

genannte Erweiterung nicht zu, was sie zur bevorzugten Methode der gemeinsamen Verwendung von SPSS und R macht.¹⁰

SPSS-Datensätze können in R mit Funktionen gelesen und geschrieben werden, die das Paket `foreign` (R Core Members, DebRoy, Bivand et al., 2014) bereitstellt. So liest die `read.spss()` Funktion `<Dateiname>.sav` Dateien.¹¹

```
> read.spss(file="<Dateiname>", use.value.labels=TRUE, to.data.frame=FALSE,
+           trim.factor.names=FALSE)
```

Variablen, deren Werte in SPSS vollständig mit labels versehen sind, konvertiert `read.spss()` in Faktoren, sofern nicht `use.value.labels=FALSE` gesetzt wird. In der Voreinstellung ist das Ergebnis ein Objekt der Klasse `list`, was durch das Argument `to.data.frame=TRUE` geändert werden kann. Mit `trim.factor.names=TRUE` wird erreicht, dass Bezeichnungen von Faktorstufen auf ihre tatsächliche Länge gekürzt werden – andernfalls können sie ungewollt 256 Zeichen umfassen. Wurden in SPSS auch labels für die Variablen vergeben, tauchen diese nach dem Import als Vektor im Attribut `variable.labels` des erstellten Objekts auf und können etwa über `attr(<Objekt>, "variable.labels")` gelesen und verändert werden.

Sollen in R bearbeitete Datensätze SPSS verfügbar gemacht werden, ist `write.foreign()` aus dem Paket `foreign` zu benutzen.

```
> write.foreign(df=<Datensatz>, datafile="<Dateiname>",
+                 codefile="<Dateiname>", package="SPSS")
```

Hierbei ist unter `datafile` der Name der Textdatei anzugeben, in der sich die eigentlichen Daten befinden sollen. Der für `codefile` einzutragende Name bezeichnet die SPSS Syntax-Datei mit der Endung `sps` mit jenen Befehlen, die in SPSS zum Einlesen dieser Daten dienen. Der erste von R in diese Datei geschriebene Befehl bezeichnet dabei den Namen der Daten-Datei – häufig empfiehlt es sich, ihm im Syntax-Editor von SPSS den vollständigen Dateipfad voranzustellen, damit SPSS die Datei in jedem Fall findet. Zudem kann SPSS Textdateien einlesen, wie sie mit `write.table(..., row.names=FALSE, sep="\t", dec=",")` erstellt werden.

Beim Import von Daten in SPSS ist darauf zu achten, dass die Variablen letztlich das richtige Format (numerisch oder Text) sowie den richtigen Skalentyp (nominal, ordinal oder metrisch) erhalten. Weiterhin orientiert sich SPSS in seiner Wahl des Dezimaltrennzeichens nicht an den Windows-Ländereinstellungen, sondern an einer internen Ländereinstellung (`locale`). Nach Möglichkeit sollten Daten also bereits mit jenem Dezimaltrennzeichen exportiert werden, das SPSS erwartet. Wo dies nicht möglich ist, lässt sich die Ländereinstellung in SPSS vor dem Import mit dem Befehl `SET LOCALE='German'`. so umschalten, dass ein Komma als Dezimaltrennzeichen gilt und mit `SET LOCALE='English'`. so, dass dies der Punkt ist. Sollten Umlaute beim Import Probleme bereiten, können sie bereits in R mit Befehlen wie `<Variable> <- gsub("ä", "æ", <Variable>)` ersetzt werden. Andernfalls erläutert die SPSS Hilfe zu den allgemeinen Optionen, wie Daten- und Syntaxdateien in unterschiedlichen Zeichencodierungen eingelesen werden können.

¹⁰Für einen detaillierten Vergleich der Arbeit mit R, SAS und SPSS vgl. Muenchen (2011), der auch den Datenaustausch zwischen diesen Programmen behandelt.

¹¹`spss.get()` aus dem Paket `Hmisc` (Harrell Jr, 2014a) verwendet `read.spss()` mit geeigneteren Voreinstellungen und verbessert den Import von Datumsangaben und Variablen-Labels.

Neben dem Datenaustausch mit SPSS gibt es auch die Möglichkeit, Daten mit Stata zu teilen, wofür ebenfalls Funktionen aus dem `foreign` Paket dienen (für Details vgl. Muenchen & Hilbe, 2010): `read.dta()` liest `<Dateiname>.dta` Dateien, `write.foreign(..., package="Stata")` schreibt sie. Der Austausch mit SAS geschieht mit Dateien im XPORT-Format analog über `read.xport()` und `write.foreign(..., package="SAS")`. Das `Hmisc` Paket stellt die Funktionen `stata.get()` und `sasxport.get()` mit geeigneteren Voreinstellungen zum Lesen von Dateien dieser Programme bereit.

Datenbanken

In R lassen sich Daten aus Datenbanken vieler verschiedener Formate direkt lesen und schreiben.¹² Dies bietet sich etwa bei extrem großen Datensätzen an, die zuviel Arbeitsspeicher belegen würden, wenn man sie als Ganzes in R öffnen wollte (vgl. Abschn. 15.3.3). Zunächst muss dafür eine Verbindung zur Datenbank hergestellt werden, woraufhin sich SQL-Kommandos wie `SELECT` in der üblichen Syntax anwenden lassen, um Daten zwischen Datenbank und R auszutauschen. Das sonst notwendige Semikolon am Ende eines SQL-Befehls ist dabei optional. Die Grundlagen von SQL erläutern die folgenden Webseiten:

<http://www.1keydata.com/sql/>
<http://sqlzoo.net/>

Für die Verbindung zu einer Datenbank kommen zwei Schnittstellen in Frage. Zum einen ist dies die ODBC-Schnittstelle (*Open DataBase Connectivity*), für die ein *client* vom Paket `RODBC` (Ripley, 2013) bereitgestellt wird. Separat für verschiedene Datenbanktypen implementieren zum anderen Zusatzpakete dieselbe Schnittstelle, die das Paket `DBI` (R Special Interest Group on Databases, 2014) definiert. Dazu zählen etwa MySQL-, PostgreSQL- oder SQLite-Datenbanken. Gegenüber der ODBC-Schnittstelle ist die DBI-basierte Schnittstelle meist leistungsfähiger.

ODBC ist plattformunabhängig und eignet sich für Datenbanken unterschiedlichen Typs sowie für Excel-Dateien. Dafür ist es auf Seiten des Betriebssystems zunächst notwendig, für die Datenbank bzw. Excel-Datei einen *data source name* (DSN) als Namen der Verbindung zu erstellen. Übernimmt dies ein Datenbank-Server nicht automatisch selbst, kann dafür unter Windows ODBC-Datenquellen aus der Gruppe *Verwaltung der Systemsteuerung* verwendet werden. Je nach eingesetzter R-Version muss die Verbindung für 32bit- oder 64bit-Clients ausgelegt sein. `vignette("RODBC")` erläutert weitere Besonderheiten, etwa für verschiedene Datenbank-Typen, und gibt Hinweise zur Installation von Treibern sowie zum Erstellen von DSNs.

Die Verwaltung der ODBC-Verbindung selbst erfolgt über Befehle mit Namen `odbc<Befehl>()`: `odbcConnect(Variante)("<DSN>")` dient zum Öffnen und `odbcClose(<DB-Verbindung>)` zum Schließen der Verbindung, `odbcGetInfo(<DB-Verbindung>)` liefert Details zur Verbindung. Für SQL-Kommandos zum Lesen und Speichern von Daten aus der verbundenen Datenbank stehen Funktionen mit Namen `sql<Befehl>()` zur Verfügung: `sqlTables(<DB-Verbindung>)` gibt eine Übersicht der vorhandenen tables aus, `sqlFetch(<DB-Verbindung>, "<table-Name>")` speichert ein table als R-Datensatz.

¹²Für eine detaillierte Beschreibung der Verwendung von Datenbanken vgl. J. Adler (2012) und Spector (2008).

Im Beispiel soll eine Datenbankverbindung zur Excel-Datei mit vorher eingetragenem DSN `data.xls` geöffnet werden, die u.a. das Tabellenblatt `sheet1` enthält. In diesem Tabellenblatt befinden sich drei Spalten mit jeweils einem Variablennamen in der ersten und Daten von fünf Beobachtungen in den folgenden Zeilen.

```
# Datenbankverbindung zu Excel-Datei im Schreib-Lese-Modus öffnen
# data.xls ist der vorher angelegte DSN
> library(RODBC)          # für odbc(Befehl)(), sql(Befehl)()
> xlsCon <- odbcConnectExcel2007("data.xls", readOnly=FALSE)
> odbcGetInfo(xlsCon)     # Verbindungsinformationen ...
> sqlTables(xlsCon)       # Tabellenblätter auflisten (Spalte TABLE_NAME)
                           TABLE_CAT TABLE_SCHEMA TABLE_NAME   TABLE_TYPE REMARKS
1 <(Pfad xls Datei)\data>      <NA>        sheet1$ SYSTEM TABLE    <NA>
2 <(Pfad xls Datei)\data>      <NA>        sheet2$ SYSTEM TABLE    <NA>
3 <(Pfad xls Datei)\data>      <NA>        sheet3$ SYSTEM TABLE    <NA>

> (myDfXls <- sqlFetch(xlsCon, "sheet1"))  # sheet1 in R speichern
   VpNr IV DV
1     1   A  4
2     2   A  6
3     3   B  7
4     4   B  8
5     5   A 11
```

In SQL-Kommandos mit `sqlQuery(<DB-Verbindung>, "<SQL-Befehl>")` ist bei Excel-Dateien zu beachten, dass den ursprünglichen Namen der Tabellenblätter im zugehörigen table-Namen ein \$ anzuhängen und der Name in [] einzuschließen ist.

```
# zeige nur bestimmte Variablen an, sortiere Ausgabe
> sqlQuery(xlsCon, "SELECT IV, DV FROM [sheet1$] ORDER BY IV")
   IV DV
1   A 11
2   A  6
3   A  4
4   B  8
5   B  7

# wähle Fälle nach verbundenen Kriterien für IV und DV aus
> sqlQuery(xlsCon, "SELECT * FROM [sheet1$] WHERE IV = 'A' AND DV < 10")
   VpNr IV DV
1     1   A  4
2     2   A  6
```

Die Datentypen, die in einer Datenbank implementiert sind, entsprechen nicht genau den in R vorhandenen. Für einen zuverlässigen Austausch sollten nur ganzzahlige oder Gleitkommazahlen sowie Zeichenketten als Daten ausgetauscht werden. Andere R-Klassen (Datumsangaben, Faktoren) sind ggf. entsprechend umzuwandeln, ehe sie mit `sqlSave(<DB-Verbindung>, <\n->Datensatz>, "<table-Name>")` gespeichert werden.

```
> myDfXls$newDV <- rnorm(nrow(myDfXls))      # Variable hinzufügen

# Datensatz in table newSheet speichern -> in Excel = neues Tabellenblatt
> sqlSave(xlsCon, myDfXls, tablename="newSheet")
> odbcClose(xlsCon)                          # DB-Verbindung schließen
```

Als Beispiel für eine Datenbank-Verbindung über die DBI-Schnittstelle soll mit dem Paket **RSQLite** (Wickham et al., 2014) zunächst eine SQLite-Datenbank neu erstellt werden, um darin einen R-Datensatz als table abzuspeichern. SQLite-Datenbanken zeichnen sich dadurch aus, dass die Datenbank eine Datei ist und kein separater Datenbank-Server lokal oder im Netzwerk laufen muss. **RSQLite** bettet den notwendigen server für eine DB-Datei ein.

Nachdem mit `dbDriver("<DB-Typ>")` ein für SQLite passendes Treiber-Objekt erzeugt wurde, kann die Verbindung zur Datenbank mit `dbConnect(<DB-Treiber>, "<DB-Name>")` hergestellt werden. Existiert die zum übergebenen Datenbank-Namen gehörende Datei noch nicht, wird sie automatisch neu angelegt.

```
> library("RSQLite")                      # für db<Funktion>()
> drv <- dbDriver("SQLite")                # SQLite-Treiber erzeugen
> con <- dbConnect(drv, "myDf.db")        # erstelle neue DB in Datei myDf.db
```

`dbWriteTable(<DB-Verbindung>, name="<table-Name>", value=<Datensatz>)` speichert den übergebenen R-Datensatz in der verbundenen Datenbank unter dem angegebenen table-Namen. `dbListTables(<DB-Verbindung>)` gibt einen Vektor mit allen table-Namen der Datenbank zurück. `dbListFields(<DB-Verbindung>, "<table-Name>")` nennt die Spalten-Namen des gewünschten Datenbank-tables.

```
# Daten simulieren
> IQ      <- rnorm(2*50, mean=100, sd=15)
> rating <- sample(LETTERS[1:3], 2*50, replace=TRUE)
> sex     <- factor(rep(c("f", "m"), times=50))
> myDf   <- data.frame(sex, IQ, rating, stringsAsFactors=FALSE)
```

```
# Datensatz myDf in table MyDataFrame der Datenbank speichern
> dbWriteTable(con, name="MyDataFrame", value=myDf, row.names=FALSE)
[1] TRUE
```

```
> dbListTables(con)                      # alle tables der Datenbank
[1] "MyDataFrame"

> dbListFields(con, "MyDataFrame")       # alle Spalten des table
[1] "sex"    "IQ"    "rating"
```

`dbReadTable(<DB-Verbindung>, "<table-Name>")` gibt das gewünschte Datenbank-table vollständig als R-Datensatz aus. `dbGetQuery(<DB-Verbindung>, "<SQL-Befehl>")` übermittelt einen SQL-Befehl und liefert das Ergebnis vollständig zurück.

```
> out <- dbReadTable(con, "MyDataFrame")  # table vollständig speichern
> head(out, n=4)                        # Kontrolle
```

```

sex      IQ rating
1   f  92.51217    A
2   m  89.28590    C
3   f 104.96750    A
4   m 102.98000    C

# SQL-Kommando für pro Gruppe berechnete Kennwerte (Mittelwert, Summe IQ)
> dbGetQuery(con, "SELECT sex, AVG(IQ) AS mIQ, SUM(IQ) AS sIQ
+                  FROM MyDataFrame GROUP BY sex")
   sex      mIQ      sIQ
1   f 101.23562 5061.781
2   m  99.34861 4967.430

```

dbSendQuery(<DB-Verbindung>, "<SQL-Befehl>") leitet den übergebenen SQL-Befehl an die Datenbank weiter und gibt ein Objekt zurück, aus dem sich mit dbFetch(<query>, n=<Anzahl>) die gewünschte Anzahl an zur Anfrage passenden Zeilen schrittweise lesen lassen. Ob alle Daten ausgelesen wurden, gibt dbHasCompleted(<query>) aus. dbClearResult(<query>) setzt die zum schrittweise Lesen offene query zurück. Dieses Vorgehen ist insbesondere bei sehr umfangreichen Ergebnissen sinnvoll. Vergleiche Abschn. 15.1.2 für die Verwendung von `while()`, um einen Befehl so oft zu wiederholen, wie eine bestimmte Nebenbedingung erfüllt ist.

```

# IQ und rating für Personen, die rating A vergeben haben
> res <- dbSendQuery(con, "SELECT IQ, rating
+                           FROM MyDataFrame WHERE rating = 'A'")

# rufe Daten schrittweise solange ab, wie noch nicht alle passenden
# Zeilen ausgegeben wurden
> while(!dbHasCompleted(res)) {
+   partial <- dbFetch(res, n=3)           # rufe 3 Zeilen ab
+   print(partial)                         # gib diese Zeilen aus
+ }
      IQ rating
1 102.80542    A
2 114.06305    A
3  91.76563    A

      IQ rating
4 120.3715    A
5 116.5732    A
6 108.8182    A                      # ...

> dbClearResult(res)                   # query zurücksetzen
[1] TRUE

```

dbRemoveTable(<DB-Verbindung>, "<table-Name>") löscht ein table in der verbundenen Datenbank, dbDisconnect(<DB-Verbindung>) beendet eine Datenbank-Verbindung. Der Rückgabewert beider Funktionen zeigt an, ob der jeweilige Befehl erfolgreich ausgeführt wurde.

```
> dbRemoveTable(con, "MyDataFrame")          # table löschen
[1] TRUE

> dbDisconnect(con)                      # DB-Verbindung trennen
[1] TRUE
```

4.2.5 Daten in der Konsole einlesen

Während es `c(<Wert1>, <Wert2>, ...)` zwar erlaubt, auf der Konsole Vektoren aus Werten zu bilden und auch ganze Datensätze einzugeben, ist dieses Vorgehen aufgrund der ebenfalls einzutippenden Kommata ineffizient. Etwas schneller ist die Dateneingabe mit dem `scan()` Befehl, bei dem nur das Leerzeichen als Trennzeichen zwischen den Daten vorhanden sein muss.

```
> scan(file="", what="numeric", na.strings="NA", dec=".")
```

Sollen Daten manuell auf der Konsole eingegeben werden, ist das Argument `file` bei der Voreinstellung `" "` zu belassen. Mit `scan()` können auch Dateien eingelesen werden, allerdings ist dies bequemer über andere Funktionen möglich (vgl. Abschn. 4.2.2). Das Argument `what` benötigt eine Angabe der Form `"logical"`, `"numeric"` oder `"character"`, die Auskunft über den Datentyp der folgenden Werte gibt. Zeichenketten müssen durch die Angabe `"character"` nicht mehr in Anführungszeichen eingegeben werden, es sei denn sie beinhalten Leerzeichen. Mit `na.strings` wird festgelegt, auf welche Weise fehlende Werte codiert sind. Das Dezimaltrennzeichen der folgenden Werte kann über das Argument `dec` definiert werden.

Beim Aufruf ist das Ergebnis von `scan()` einem Objekt zuzuweisen, damit die folgenden Daten auch gespeichert werden. Auf den Befehlsaufruf `scan()` hin erscheint in der Konsole eine neue Zeile als Signal dafür, dass nun durch Leerzeichen getrennt Werte eingegeben werden können. Eine neue Zeile wird dabei durch Drücken der `Return` Taste begonnen und zeigt in der ersten Spalte an, der wievielte Wert folgt. Die Eingabe der Werte gilt als abgeschlossen, wenn in einer leeren Zeile die `Return` Taste gedrückt wird.

```
> vec <- scan()
1: 123 456 789
4:
Read 3 items

> charVec <- scan(what="character")
1: as df ej kl
5:
Read 4 items
```

4.2.6 Unstrukturierte Textdateien

Nicht immer liegen einzulesende Daten bereits in Form einer Tabelle vor, für die sich `read.table()` eignet. Mit `readLines("<Pfad>", n=<Anzahl>)` können auch unstrukturierte Daten im Textformat eingelesen werden, die noch nicht in rechteckiger Form organisiert sind. Das Ergebnis ist ein

Vektor aus Zeichenketten, jedes dessen Elemente jeweils eine Zeile der Datei speichert. Die einzelnen Zeilen können dann mit Methoden zur Manipulation von Zeichenketten weiterverarbeitet werden, die Abschn. 2.12 vorstellt.

Als Besonderheit lässt sich über das Argument `n` von `readLines()` auch steuern, wie viele Zeilen der Datei gleichzeitig gelesen werden sollen – in der Voreinstellung die gesamte Datei. Setzt man etwa `n=5`, umfasst die Ausgabe des ersten Aufrufs die Zeilen 1–5 in Form von fünf Elementen eines Vektors aus Zeichenketten. Dabei merkt sich `readLines()` die Position, an der zuletzt gelesen wurde und setzt beim nächsten Aufruf an dieser Position fort. Ein erneuter Aufruf mit `n=4` würde also die Zeilen 6–9 zurückliefern. Analog speichert `writeLines(<Vektor>, <Datei>)` die Inhalte eines Vektors aus Zeichenketten in die angegebene Datei, wobei jedes Element des Vektors in eine separate Zeile geschrieben wird.

4.2.7 Datenqualität sicherstellen

Nachdem Daten in R importiert wurden, sollten Sie auf ihre Qualität geprüft werden. Wichtige Kriterien einer hohen Datenqualität sind folgende:

- Einheitlichkeit der Codierung, insbesondere bei
 - Angaben zu Datum oder Uhrzeit (vgl. Abschn. 2.13): Aufgrund der Vielzahl national wie international unterschiedlicher Formatierungsmöglichkeiten sind diese Variablen beim Datenaustausch besonders fehlerträchtig.
 - Eigennamen: Vor allem Umlaute, Bindestriche bei Doppelnamen, mehrere Vornamen, Namenszusätze wie „de“, „von“ sowie Groß- und Kleinschreibung können dafür sorgen, dass Probleme beim Datenimport oder beim Zusammenführen mehrerer Datensätze mit Beobachtungen derselben Personen auftreten. Sie lassen sich mit Methoden zur Verarbeitung von Zeichenketten lösen, die in Abschn. 2.12 beschrieben sind. Für die Identifizierung fast genau passender Zeichenketten vgl. die dortige Fußnote 49.
 - Physikalischen Variablen: Hier ist bei der Integration mehrerer Datensätze darauf zu achten, dass dieselben physikalischen Einheiten verwendet werden.
- Vollständigkeit: Werden etwa Datensätze mit `merge()` zusammengeführt (vgl. Abschn. 3.3.7), besteht die Gefahr, dass aufgrund uneinheitlicher Codierung Einträge fälschlicherweise nicht als übereinstimmend gewertet und deshalb Personen vollständig gelöscht werden. Deshalb sollte nach Möglichkeit anhand der Menge der eindeutigen Personen-IDs sichergestellt werden, dass keine Personen bei der Datenaufbereitung verloren gehen – z. B. mit `all(<IDs vor> %in% <IDs nach>)`.
- Richtigkeit:
 - Falsche Werte können etwa aus einer fehlerhaften Messung oder aus Tippfehlern bei der Eingabe herrühren. Deshalb sollte die Verteilung aller Variablen mit deskriptiven Kennwerten ebenso wie mit Diagrammen auf ihre Plausibilität geprüft werden. Relevant sind dabei z. B. die Verteilungsform, Werte außerhalb des möglichen Messbereichs und Ausreißer. Vergleiche Abschn. 2.7 für die deskriptive Beschreibung

kontinuierlicher Größen sowie Abschn. 14.6 und 14.8 für Möglichkeiten, ihre (gemeinsame) Verteilung in Diagrammen zu veranschaulichen. Kategoriale Variablen lassen sich durch ihre (gemeinsamen) Häufigkeitsverteilungen mit den in Abschn. 2.10 erläuterten Mitteln charakterisieren. Die zugehörigen Säulendiagramme sind in Abschn. 14.4 beschrieben. Abschnitt 6.5 zeigt, wie Extremwerte und Ausreißer im Rahmen der Regressionsdiagnostik identifiziert werden können.

- Fehlende Werte werden in verschiedenen Programmen uneinheitlich codiert, etwa mit besonderen Zahlen wie 999. Fehlende Werte müssen in R auf `NA` umcodiert werden, damit sie nicht fälschlicherweise in die Auswertung einbezogen werden (vgl. Abschn. 2.11).
- Eindeutigkeit: Ob beim Zusammenführen von Daten aus mehreren Quellen doppelte Fälle auftreten, kann wie in Abschn. 3.3.4 gezeigt untersucht und ggf. behoben werden.

4.3 Dateien verwalten

R verfügt über eine Reihe von Funktionen, die Dateien finden und verändern können, die zum Import oder Export von Daten benötigt werden. Dabei spielen Pfadangaben zu Ordnern und Dateien eine zentrale Rolle.

Obwohl unter Windows üblicherweise der *backslash* \ in Pfadangaben als Verzeichnistrenner dient, sollte er in R nicht verwendet werden. Stattdessen ist bei allen Pfadangaben wie unter MacOS und Linux bevorzugt der *forward slash* / zu benutzen, etwa "c:/work/r/datei.txt". Alternativ ist weiterhin der doppelte backslash \\ möglich: "c:\\work\\r\\datei.txt". Pfade können entweder relativ zum aktuellen, von `getwd()` ausgegebenen Arbeitsverzeichnis sein ("path/file.txt"), oder absolut, d. h. wie beim vorherigen Beispiel beginnend mit dem Laufwerksbuchstaben (Windows) oder dem Stammverzeichnis "/" (MacOS, Linux).

4.3.1 Dateien auswählen

Die Pfadangabe einer Datei lässt sich entweder interaktiv oder über ein vorgegebenes Suchmuster bestimmen. Den Namen einer einzelnen Datei erhält man interaktiv über ein Dialogfeld zur Dateiauswahl, das durch `file.choose()` aufgerufen wird. Die Funktion gibt den Namen der ausgewählten Datei inkl. des vollständigen Pfades zurück. Eine Variante, die gleichzeitig mehrere Dateien auswählen lässt, existiert nur unter Windows. Dort gibt `choose.files()` einen Vektor von Pfadnamen zu den ausgewählten Dateien zurück.

Mit `list.files()` können alle Dateien in einem Ordner aufgelistet werden, deren Name zu einem Suchmuster passt.

```
> list.files("~/Ordner", pattern="Suchmuster", full.names=FALSE,  
+           recursive=FALSE, ignore.case=FALSE)
```

Als erstes Argument ist der Pfad zu einem Ordner anzugeben. `pattern` akzeptiert ein Suchmuster in Form eines regulären Ausdrucks (vgl. Abschn. 2.12.4), das sich für Dateinamen häufig einfacher mit `glob2rx()` über Platzhalter wie * formulieren lässt (vgl. Abschn. 2.12.5). In

der Voreinstellung erhält man nur die Namen der Dateien im angegebenen Ordner, die zum Suchmuster passen. Benötigt man den vollständigen Pfad zur Datei, ist `full.names=TRUE` zu setzen. Die Argumente `recursive` und `ignore.case` bestimmen jeweils, ob auch Unterordner durchsucht bzw. ob Groß- und Kleinschreibung im Dateinamen beim Abgleich mit dem Suchmuster ignoriert werden sollen.

```
# vollständige Pfadnamen für alle Dateien im Verzeichnis d:/files
# mit der Endung .txt
> paths <- list.files(path="d:/files", pattern="\.\.txt$", full.names=TRUE)

# lies diese Dateien in eine Liste aus Datensätzen ein
> DFlist <- lapply(paths, function(f) {
+   read.table(f, header=TRUE, stringsAsFactors=FALSE) })

# verbinde alle Datensätze in der Liste zu einem gemeinsamen Datensatz
> DFall <- do.call("rbind", DFlist)
```

Alternativ erhält man mit `Sys.glob("<Pfad>")` aus einem vorgegebenen Pfad einen Vektor von Dateinamen, die auf ein bestimmtes Muster passen. Die Pfadangabe gibt sowohl den Pfad als auch das Muster für die Dateinamen vor, wobei bestimmte Platzhalter erlaubt sind: * steht für eine beliebige Zeichenfolge, ? für ein einzelnes beliebiges Zeichen, . für das aktuelle Verzeichnis, .. für das Verzeichnis eine Ebene über dem aktuellen Verzeichnis und ~ für das Heimverzeichnis des Benutzers. Ob die zurückgegebenen Dateipfade absolut oder relativ zum aktuellen Arbeitsverzeichnis sind, orientiert sich am übergebenen Suchmuster.

```
# alle Dateien mit Endung .tex im Parallelordner gddmr_tex
# relativer Pfad zum derzeitigen Arbeitsverzeichnis
> texFiles1 <- Sys.glob("../gddmr_tex/*.tex")
> head(texFiles1, n=4)
[1] "../gddmr_tex/gddmr.tex"           ".../gddmr_tex/rl_00_preface.tex"
[3] "../gddmr_tex/rl_00_settings.tex"  ".../gddmr_tex/rl_01_first_steps.tex"

# absoluter Pfad
> texFiles2 <- Sys.glob("d:/work/gddmr_tex/*.tex")
> head(texFiles2, n=2)
[1] "d:/work/gddmr_tex/r_long_tex/gddmr.tex"
[2] "d:/work/gddmr_tex/r_long_tex/rl_00_preface.tex"
```

4.3.2 Dateipfade manipulieren

In der vollständigen Pfadangabe zu einer Datei steht einerseits die Information zum Ordner, der die Datei enthält, andererseits die Information zum Dateinamen sowie ggf. ihrer Endung. Diese Informationen lassen sich getrennt aus einem vollständigen Pfad extrahieren.

So gibt `basename("<Dateipfad>")` nur den Dateinamen aus, entfernt also die Angaben zum Ordner aus der übergebenen Pfadangabe. Als Komplement nennt `dirname("<Dateipfad>")`

den Pfad zum Ordner, der eine Datei enthält, entfernt also den Dateinamen. Nur die Dateiendung (ohne Ordner und Dateinamen) erhält man mit `tools::file_ext("Dateipfad")`. Als Komplement gibt `tools::file_path_sans_ext("Dateipfad")` den vollständigen Pfad mit Dateinamen, aber ohne Endung zurück.¹³

```
> basename("c:/path/to/file.txt")
[1] "file.txt"

> dirname("c:/path/to/file.txt")
[1] "c:/path/to"

> tools::file_ext("c:/path/to/file.txt")
[1] "txt"

> tools::file_path_sans_ext("c:/path/to/file.txt")
[1] "c:/path/to/file"
```

4.3.3 Dateien verändern

Die meisten Dateioperationen, die man interaktiv mit einem Dateimanager vornimmt, können mit den folgenden Funktionen automatisiert werden:

- `dir.exists("<Ordnerpfad>")` prüft, ob der Ordner mit dem angegebenen Pfad bereits existiert und tatsächlich ein Ordner ist.
- `file.exists("<Dateipfad>")` prüft, ob die Datei mit dem angegebenen Pfad bereits existiert.
- `file.remove("<Dateipfad>")` löscht die Datei mit dem angegebenen Pfad.
- `file.copy(from="<Dateipfad>", to="<Dateipfad>")` erstellt die Kopie einer Datei `from` an der Stelle `to`.
- `file.rename(from="<Dateipfad>", to="<Dateipfad>")` verschiebt eine Datei `from` an die Stelle `to`.
- `file.create("<Dateipfad>")` erstellt eine leere Datei mit dem übergebenen Pfad.
- `dir.create("<Ordnerpfad>")` erstellt einen Ordner mit dem übergebenen Pfad.

Mit dem Rückgabewert TRUE oder FALSE melden die Funktionen, ob eine Dateioperation erfolgreich durchgeführt werden konnte.

```
# neues Verzeichnis newDir relativ zum aktuellen Arbeitsverzeichnis
> dir.create("newDir")
> file.create("newDir/newFile.txt")           # neue Datei in newDir
[1] TRUE
```

¹³Die beiden letztgenannten Funktionen stammen aus dem im Basisumfang von R enthaltenen Paket `tools`, das diese Funktionen jedoch nicht öffentlich macht und deswegen dem Funktionsnamen explizit vorangestellt werden muss (vgl. Abschn. 15.3.1).

```
# Kopie der neu erstellten Datei
> file.copy("newDir/newFile.txt", to="newDir/fileA.txt")
[1] TRUE

# benenne Kopie um
> file.rename("newDir/fileA.txt", to="newDir/fileB.txt")
[1] TRUE

> file.remove("newDir/newFile.txt")          # entferne ursprüngliche Datei
[1] TRUE

> file.exists("newDir/newFile.txt")          # existiert ursprüngliche Datei?
[1] FALSE

> file.exists("newDir/fileB.txt")            # existiert umbenannte Kopie?
[1] TRUE
```

Kapitel 5

Hilfsmittel für die Inferenzstatistik

Bevor in den kommenden Kapiteln Funktionen zur inferenzstatistischen Datenanalyse besprochen werden, ist es notwendig Hilfsmittel vorzustellen, auf die viele dieser Funktionen zurückgreifen. Dazu gehören die Syntax zur Formulierung linearer Modelle sowie einige Familien statistischer Verteilungen von Zufallsvariablen, die bereits bei der Erstellung zufälliger Werte in Erscheinung getreten sind (vgl. Abschn. 2.4.4). Zunächst ist die Bedeutung wichtiger inhaltlicher Begriffe zu klären, die im Kontext inferenzstatistischer Tests häufig auftauchen.

5.1 Wichtige Begriffe inferenzstatistischer Tests

Die Terminologie bei der Darstellung inferenzstatistischer Tests ist in der Literatur nicht einheitlich, deswegen soll der folgende Abschnitt kurz präzisieren, mit welcher Bedeutung zentrale Begriffe im weiteren Verlauf des Buches verwendet werden. Die inhaltlichen Zusammenhänge der Logik schließender Statistik seien dabei als bekannt vorausgesetzt (Eid et al., 2013).

Die *Nullhypothese*, deren Konsistenz mit beobachteten Daten ein Test prüft, soll im folgenden mit H_0 abgekürzt werden, die *Alternativhypothese* entsprechend mit H_1 . Die *Teststatistik* ist eine Zufallsvariable, deren Verteilung unter Gültigkeit der H_0 mit spezifischen Zusatzannahmen bekannt ist. Die Wahrscheinlichkeit dafür, dass sie Werte in beliebigen Intervallen annimmt, lässt sich dann über ihre Verteilungsfunktion berechnen. Theoretische Parameter einer Verteilung sollen i. d. R. mit griechischen Buchstaben (etwa μ für den Erwartungswert), empirische Kennwerte mit lateinischen Buchstaben benannt werden (etwa M für den Mittelwert). Besitzen empirische Schätzer keinen eigenen lateinischen Buchstaben, wird für sie der griechische Buchstabe des zu schätzenden Parameters mit einem Circumflex versehen (etwa $\hat{\epsilon}$). Im Interesse einfacherer Formulierungen wird im folgenden nicht streng zwischen einer empirischen Variable und der zugehörigen Zufallsvariable im statistischen Sinn unterschieden.

Die Wahrscheinlichkeit unter Gültigkeit der H_0 , dass die Teststatistik Werte annimmt, die i. S. der H_1 mindestens so extrem wie der beobachtete sind, heißt *p-Wert*.¹ Der *kritische Wert* eines Tests soll den Wert bezeichnen, den die Teststatistik überschreiten muss, damit die Entscheidung für die H_1 ausfällt. Abweichend davon wird in der Literatur auch der mindestens zu erreichende Wert als der kritische bezeichnet, also der erste Wert des Ablehnungsbereichs der H_0 . Dieser Unterschied in der Bezeichnungskonvention ist nur für diskrete Verteilungen relevant.

¹Handelt es sich um eine zusammengesetzte H_0 – etwa bei gerichteter H_1 , ist hier immer die H_0 gemeint, die am dichtesten an der H_1 liegt.

Der Fehler, sich bei tatsächlicher Gültigkeit der H_0 für die H_1 zu entscheiden, ist der *Fehler erster Art* oder auch α -*Fehler*. Der Fehler, die H_0 bei tatsächlicher Gültigkeit der H_1 nicht zu verwerfen, wird als β -*Fehler* bezeichnet. Wenn zur sprachlichen Vereinfachung von der Größe oder Höhe eines Fehlers die Rede ist, soll immer die Wahrscheinlichkeit für das Eingehen eines solchen Fehlers gemeint sein. Als α -*Niveau* (auch einfach: α) wird die maximale Wahrscheinlichkeit eines Fehlers erster Art bezeichnet, die man bei Konstruktion einer Entscheidungsregel für den Test (Wahl des kritischen Wertes) einzugehen bereit ist. Ein statistischer Test fällt dann signifikant aus, wenn der p -Wert kleiner als das gesetzte α -Niveau ist. Die *power* oder *Teststärke* eines Tests ist die Wahrscheinlichkeit unter Gültigkeit der H_1 , dass die Teststatistik Werte größer als der kritische Wert annimmt.²

Mit dem *Vertrauensintervall* (VI) bzw. *Konfidenzintervall* zur Wahrscheinlichkeit $1 - \alpha$ für einen theoretischen Parameter θ ist ein nach unten wie oben offenes Intervall (a, b) gemeint: Die durch eine konkrete Formel zu spezifizierende Konstruktionsmethode liefert mit einer Wahrscheinlichkeit von $1 - \alpha$ Grenzen a und b , so dass θ im zugehörigen Intervall liegt ($a < \theta < b$). Ein statistischer Test zum Niveau α ist genau dann signifikant, wenn das zugehörige $1 - \alpha$ Vertrauensintervall den Wert des theoretischen Parameters unter H_0 nicht enthält.

Wenn sich ein Test sowohl gerichtet als auch ungerichtet durchführen lässt, bietet R beim Aufruf der Auswertungsfunktion eine Option zur Angabe, ob es sich um eine zweiseitige bzw. um welche einseitige H_1 (links- oder rechtsseitig) es sich handelt. Der ausgegebene p -Wert berücksichtigt die Art der Fragestellung und ist deshalb immer direkt mit dem gewählten α -Niveau zu vergleichen. Genauso werden Konfidenzintervalle für geschätzte Parameter entsprechend der Richtung der Fragestellung berechnet: Im Fall einer zweiseitigen Fragestellung wird das zweiseitige, im Fall einer einseitigen Fragestellung entsprechend das passende einseitige Konfidenzintervall gebildet.

5.2 Lineare Modelle formulieren

Manche Funktionen in R erwarten als Argument die symbolische Formulierung eines linearen statistischen Modells, dessen Passung für die zu analysierenden Daten getestet werden soll. Eine solche Modellformel besitzt die Klasse `formula` und beschreibt, wie der systematische Anteil von Werten einer Zielvariable (abhängige Variable, AV) aus Werten einer oder mehrerer Prädiktoren (unabhängige Variablen, UVn) theoretisch hervorgeht.³ Die Annahme der prinzipiellen Gültigkeit eines linearen Modells über das Zustandekommen von Variablenwerten steht hinter vielen statistischen Verfahren, etwa der linearen Regression, Varianz- oder Kovarianzanalyse. Ein Formelobjekt wird wie folgt erstellt:

```
> modellierte Variable ~ lineares Modell
```

²In dieser Darstellung werden die unterschiedlichen Ansätze von Fisher sowie Neyman und Pearson vermischt. Während für Fisher der p -Wert entsprechend des Likelihood-Prinzips ein Maß für die lokale Evidenz vorliegender Daten gegen die H_0 ist, begrenzen Neyman und Pearson die langfristige Fehlerrate erster Art der häufig angewendeten Entscheidungsregel für die Wahl zwischen H_0 und H_1 auf α .

³Modellformeln verwenden die *Wilkinson-Rogers-Notation* (G. N. Wilkinson & Rogers, 1973; für Details vgl. `?formula`). Im Sinne des allgemeinen linearen Modells beschreibt die rechte Seite einer Modellformel die spaltenweise Zusammensetzung der Designmatrix, die `model.matrix(Modelformel)` für ein konkretes Modell ausgibt (vgl. Abschn. 12.9 und Venables & Ripley, 2002, p. 144 ff.). Bei einem multivariaten Modell können auch mehrere AVn vorhanden sein.

Links der Tilde \sim steht die Variable, deren systematischer Anteil sich laut Modellvorstellung aus anderen Variablen ergeben soll. Die modellierenden Variablen werden in Form einzelner Terme rechts der \sim aufgeführt. Im konkreten Fall werden für alle Terme die Namen von Datenvektoren oder Faktoren derselben Länge eingesetzt.

Im Modell der einfachen linearen Regression (vgl. Kap. 6) sollen sich etwa die Werte des Kriteriums aus Werten des quantitativen Prädiktors ergeben, hier hat die Modellformel also die Form $\langle \text{Kriterium} \rangle \sim \langle \text{Prädiktor} \rangle$. In der Varianzanalyse (vgl. Abschn. 7.3) hat die AV die Rolle der modellierten und die kategorialen UVn die Rolle der modellierenden Variablen. Hier hat die Modellformel die Form $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$. Um R die Möglichkeit zu geben, beide Fälle zu unterscheiden, müssen im Fall der Regression die Prädiktoren numerische Vektoren sein, die UVn in der Varianzanalyse dagegen Objekte der Klasse **factor**.

Es können mehrere, in der Modellformel durch $+$ getrennte Vorhersageterme in ein statistisches Modell eingehen. Ein einzelner Vorhersageterm kann dabei entweder aus einer Variable oder aber aus der Kombination von Variablen i.S. ihrer statistischen Interaktion bestehen. Die Beziehung zwischen den zu berücksichtigenden Variablen wird durch Symbole ausgedrückt, die sonst numerische Operatoren darstellen, rechts der \sim in einer Modellformel aber eine andere Bedeutung tragen. An Möglichkeiten, Variablen in einem Modell zu berücksichtigen, gibt es u.a. die in Tab. 5.1 aufgeführten.

Tabelle 5.1: Notation für die Modellformel linearer Modelle

Operator	übliche Bedeutung	Bedeutung in einer Modellformel
$+$	Addition	den folgenden Vorhersageterm hinzufügen
$-$	Subtraktion	den folgenden Vorhersageterm ausschließen
$\langle A \rangle : \langle B \rangle$	Sequenz	Interaktion $A \times B$ als Vorhersageterm
$\langle A \rangle * \langle B \rangle$	Multiplikation	Kurzform für $A + B + A:B$ (alle additiven und Interaktionseffekte)
\sim	potenzieren	Begrenzung des Grads zu berücksichtigender Interaktionen
$\langle A \rangle \%in\% \langle B \rangle$	Element von Menge	bei Verschachtelung von A in B (genestetes Design): Interaktion $A \times B$ als Vorhersageterm
$\langle A \rangle / \langle B \rangle$	Division	bei Verschachtelung von A in B (genestetes Design): Kurzform für $A + A \%in\% B$
1	1	absoluter Term (Gesamterwartungswert). Implizit vorhanden, wenn nicht durch -1 ausgeschlossen
$.$		bei Verwendung eines Datensatzes: alle vorhandenen Variablen, bis auf die bereits explizit verwendeten
$.$		bei Veränderung eines Modells: alle bisherigen Terme

Als Beispiel gebe es eine kontinuierliche AV Y , drei quantitative Prädiktoren X_1, X_2, X_3 sowie zwei Faktoren F_1, F_2 . Neben den additiven Effekten der Terme können auch ihre Interaktionseffekte berücksichtigt werden. Zudem beinhaltet das Modell i. d. R. einen absoluten Term (den y -Achsenabschnitt der Vorhersagegerade im Fall der einfachen linearen Regression), der aber auch unterdrückt werden kann. Mit der Kombination von quantitativen Prädiktoren und Faktoren können etwa folgende lineare Modelle spezifiziert werden.

```
> Y ~ X1                      # einfache lineare Regression von Y auf X1
> Y ~ X1 + X2 - 1             # multiple lineare Regression von Y auf X1 und X2
                             # ohne absoluten Term (y-Achsenabschnitt)
> Y ~ F1                      # einfaktorielle Varianzanalyse
> Y ~ F1 + F2 + F1:F2        # zweifaktorielle Varianzanalyse
                             # mit beiden Haupteffekten und der Interaktion
> Y ~ X1 + F1                 # Kovarianzanalyse mit Kovariate X1 und Faktor F1
> Y ~ X1*X2                   # multiple lineare Regression von Y auf X1 und X2
                             # sowie auf den Interaktionsterm von X1 und X2
> Y ~ (X1 + X2 + X3)^2       # Regression von Y auf alle additiven Effekte
                             # sowie alle Interaktionseffekte bis zum 1. Grad:
                             # Y ~ X1 + X2 + X3 + X1:X2 + X1:X3 + X2:X3
```

Die sich ergebenden Vorhersageterme einer Modellformel sowie weitere Informationen zum Modell können mit der Funktion `terms(⟨Modellformel⟩)` erfragt werden, deren Ausgabe hier gekürzt ist.

```
> terms(Y ~ X1*X2*X3)          # ...
[1] "X1" "X2" "X3" "X1:X2" "X1:X3" "X2:X3" "X1:X2:X3"

> terms(Y ~ (X1 + X2 + X3)^2 - X1 - X2:X3)          # ...
[1] "X2" "X3" "X1:X2" "X1:X3"
```

Innerhalb einer Modellformel können die Terme selbst das Ergebnis der Anwendung von Funktionen auf Variablen sein. Soll etwa nicht Y als Kriterium durch X als Prädiktor vorhergesagt werden, sondern der Logarithmus von Y durch den Betrag von X , lautet die Modellformel $\log(Y) \sim \text{abs}(X)$. Sollen hierbei innerhalb einer Modellformel Operatoren in ihrer arithmetischen Bedeutung zur Transformation von Variablen verwendet werden, muss der entsprechende Term in `I(⟨Transformation⟩)` eingeschlossen werden. Um etwa das Doppelte von X als Prädiktor für Y zu verwenden, lautet die Modellformel damit $Y \sim I(2*X)$.

5.3 Funktionen von Zufallsvariablen

Mit R lassen sich die Werte von häufig benötigten Dichte- bzw. Wahrscheinlichkeitsfunktionen, Verteilungsfunktionen und deren Umkehrfunktionen an beliebiger Stelle bestimmen.⁴ Dies erübrigt es, etwa für p -Werte oder kritische Werte Tabellen konsultieren und bei nicht tabellierten Werten für Wahrscheinlichkeiten oder Freiheitsgrade zwischen angegebenen Werten interpolieren

⁴Für weitere Verteilungen vgl. `?Distributions` sowie den Abschnitt *Probability Distributions* der CRAN Task Views (Dutang, 2014).

zu müssen. Tabelle 5.2 gibt Auskunft über einige der hierfür verfügbaren Funktionsfamilien sowie über ihre Argumente und deren Voreinstellungen. Für ihre Verwendung zur Erzeugung von Zufallszahlen vgl. Abschn. 2.4.4.

Tabelle 5.2: Vordefinierte Funktionen von Zufallsvariablen

Familienname	Funktion	Argumente mit Voreinstellung
binom	Binomialverteilung	size, prob
chisq	χ^2 -Verteilung	df, ncp=0
exp	Exponentialverteilung	rate=1
f	F -Verteilung	df1, df2, ncp=0
gamma	Γ -Funktion	shape, rate=1, scale=1/rate
hyper	Hypergeometrische Verteilung	m, n, k
logis	Logistische Verteilung	location=0, scale=1
multinom	Multinomialverteilung	size, prob
norm	Normalverteilung ⁵	mean=0, sd=1
pois	Poisson-Verteilung	lambda
signrank	Wilcoxon-Vorzeichen-Rangverteilung	x, n
t	t -Verteilung	df, ncp=0
unif	Gleichverteilung	min=0, max=1
weibull	Weibull-Verteilung	shape, scale=1
wilcox	Wilcoxon-Rangsummenverteilung	m, n

5.3.1 Dichtefunktion

Mit Funktionen, deren Namen nach dem Muster `d(Funktionsfamilie)` aufgebaut sind, lassen sich die Werte der Dichtefunktionen⁶ der in Tab. 5.2 genannten Funktionsfamilien bestimmen. Mit dem Argument `x` wird angegeben, für welche Stelle der Wert der Dichtefunktion berechnet werden soll. Dies kann auch ein Vektor sein – dann wird für jedes Element von `x` der Wert der Dichtefunktion bestimmt. Die Bedeutung der übrigen Argumente ist identisch zu jener bei den zugehörigen Funktionen zum Generieren von Zufallszahlen (vgl. Abschn. 2.4.4).

```
> dbinom(x, size, prob)                                # Binomialverteilung
> dnorm(x, mean=0, sd=1)                             # Normalverteilung
> dchisq(x, df,          ncp=0)                      # chi^2-Verteilung
> dt(x, df,          ncp=0)                           # t-Verteilung
```

⁵Für multivariate t - und Normalverteilungen vgl. das Paket `mvtnorm` (Genz et al., 2014; Genz & Bretz, 2009).

⁶Im Fall diskreter (z. B. binomialverteilter) Variablen die Wahrscheinlichkeitsfunktion. Zusätzlich existiert mit `pbirthday()` eine Funktion zur Berechnung der Wahrscheinlichkeit, dass in einer Menge mit `n` Elementen `coincident` viele denselben Wert auf einer kategorialen Variable mit `classes` vielen, gleich wahrscheinlichen Stufen haben. Mit `classes=365` und `coincident=2` ist dies die Wahrscheinlichkeit, dass zwei Personen am selben Tag Geburtstag haben.

```
> df(x, df1, df2, ncp=0) # F-Verteilung
```

Die Wahrscheinlichkeit, beim zehnfachen Werfen einer fairen Münze genau siebenmal Kopf als Ergebnis zu erhalten, ergibt sich beispielsweise so:

```
> dbinom(7, size=10, prob=0.5)
[1] 0.1171875
```

```
> choose(10, 7) * 0.5^7 * (1-0.5)^(10-7) # manuelle Kontrolle
[1] 0.1171875
```

5.3.2 Verteilungsfunktion

Die Werte der zu einer Dichte- bzw. Wahrscheinlichkeitsfunktion gehörenden Verteilungsfunktionen lassen sich mit Funktionen berechnen, deren Namen nach dem Muster `p(Funktionsfamilie)` aufgebaut sind. Mit dem Argument `q` wird angegeben, für welche Stelle der Wert der Verteilungsfunktion berechnet werden soll. In der Voreinstellung sorgt das Argument `lower.tail=TRUE` dafür, dass der Rückgabewert an einer Stelle q die Wahrscheinlichkeit angibt, dass die zugehörige Zufallsvariable Werte $\leq q$ annimmt. Die Gegenwahrscheinlichkeit (Werte $> q$) wird mit dem Argument `lower.tail=FALSE` berechnet.⁷

```
> pbinom(q, size, prob, lower.tail=TRUE)      # Binomialverteilung
> pnorm(q, mean=0, sd=1, lower.tail=TRUE)      # Normalverteilung
> pchisq(q, df, ncp=0, lower.tail=TRUE)        # chi^2-Verteilung
> pt(q, df, ncp=0, lower.tail=TRUE)            # F-Verteilung
> pf(q, df1, df2, ncp=0, lower.tail=TRUE)       # F-Verteilung

> pbinom(7, size=10, prob=0.5)    # Verteilungsfunktion Binomialverteilung
[1] 0.9453125

> sum(dbinom(0:7, size=10, prob=0.5))         # Kontrolle über W-Funktion
[1] 0.9453125

> pnorm(c(-Inf, 0, Inf), mean=0, sd=1)        # Standardnormalverteilung
[1] 0.0 0.5 1.0

# Standardnormalverteilung: Fläche unter Dichtefunktion rechts von 1.645
> pnorm(1.645, mean=0, sd=1, lower.tail=FALSE)
[1] 0.04998491

# äquivalent: 1-(Fläche unter Dichtefunktion links von 1.645)
> 1-pnorm(1.645, mean=0, sd=1, lower.tail=TRUE)
```

⁷Bei der Verwendung von Verteilungsfunktionen diskreter (z. B. binomialverteilter) Variablen ist zu beachten, dass die Funktion die Wahrscheinlichkeit dafür berechnet, dass die zugehörige Zufallsvariable Werte $\leq q$ annimmt – die Grenze q also mit eingeschlossen ist. Für die Berechnung der Wahrscheinlichkeit, dass die Variable Werte $\geq q$ annimmt, ist als erstes Argument deshalb $q - 1$ zu übergeben, andernfalls würde nur die Wahrscheinlichkeit für Werte $> q$ bestimmt.

```
[1] 0.04998491
```

Mit der Verteilungsfunktion lässt sich auch die Wahrscheinlichkeit dafür berechnen, dass die zugehörige Variable Werte innerhalb eines bestimmten Intervalls annimmt: Dazu ist der Wert der unteren Intervallgrenze von jenem der oberen zu subtrahieren.

```
# Standardnormalverteilung: Wkt. für Werte im Intervall mu +- sd
> m <- 100                                # Erwartungswert
> s <- 15                                    # Standardabweichung
> diff(pnorm(c(m-s, m+s), mean=m, sd=s))
[1] 0.6826895
```

Nützlich ist die Verteilungsfunktion insbesondere für die manuelle Berechnung des p -Wertes in inferenzstatistischen Tests: Ist q der Wert einer stetigen Teststatistik, liefert $1 - p(Familie)(q, \dots)$ ebenso den zugehörigen p -Wert des rechtsseitigen Tests wie man ihn auch direkt mit $p(Familie)(q, \dots, lower.tail=FALSE)$ erhalten kann (vgl. Fußnote 7).

5.3.3 Quantilfunktion

Die Werte der zu einer Dichte- bzw. Wahrscheinlichkeitsfunktion gehörenden Quantilfunktion lassen sich mit Funktionen berechnen, deren Namen nach dem Muster `q(Funktionsfamilie)` aufgebaut sind. Mit dem Argument `p` wird angegeben, für welche Wahrscheinlichkeit der Quantilwert berechnet werden soll. Das Ergebnis ist die Zahl, die in der zugehörigen Dichtefunktion die Fläche p links (Argument `lower.tail=TRUE`) bzw. rechts (`lower.tail=FALSE`) abschneidet. Anders formuliert ist das Ergebnis der Wert, für den die zugehörige Verteilungsfunktion den Wert p annimmt.⁸ Die Quantilfunktion ist also die Umkehrfunktion der Verteilungsfunktion.

```
> qbinom(p, size, prob,      lower.tail=TRUE)      # Binomialverteilung
> qnorm(p, mean=0, sd=1,     lower.tail=TRUE)      # Normalverteilung
> qchisq(p, df,            ncp=0, lower.tail=TRUE) # chi^2-Verteilung
> qt(p, df,               ncp=0, lower.tail=TRUE) # t-Verteilung
> qf(p, df1, df2, ncp=0, lower.tail=TRUE)        # F-Verteilung
```

Die Quantilfunktion lässt sich nutzen, um kritische Werte für inferenzstatistische Tests zu bestimmen. Dies erübrigt es Tabellen zu konsultieren und die damit verbundene Notwendigkeit zur Interpolation bei nicht tabellierten Werten für Wahrscheinlichkeiten oder Freiheitsgrade.

```
> qnorm(pnorm(0))          # qnorm() ist Umkehrfunktion von pnorm()
[1] 0

> qnorm(1-(0.05/2), 0, 1) # krit. Wert zweiseitiger z-Test, alpha=0.05
[1] 1.959964

# krit. Wert zweiseitiger z-Test, alpha=0.05
```

⁸Bei diskreten Verteilungen (z. B. Binomialverteilung) ist das Ergebnis bei `lower.tail=TRUE` der kleinste Wert, der in der zugehörigen Wahrscheinlichkeitsfunktion mindestens p links abschneidet. Bei `lower.tail=FALSE` ist das Ergebnis entsprechend der größte Wert, der mindestens p rechts abschneidet.

Kapitel 5 Hilfsmittel für die Inferenzstatistik

```
> qnorm(0.05/2, 0, 1, lower.tail=FALSE)
[1] 1.959964

# krit. Wert einseitiger t-Test, alpha=0.01, df=18
> qt(0.01, 18, 0, lower.tail=FALSE)
[1] 2.552380
```

Kapitel 6

Lineare Regression

Die Korrelation zweier quantitativer Variablen ist ein Maß ihres linearen Zusammenhangs. Auch die lineare Regression analysiert den linearen Zusammenhang von Variablen, um die Werte einer Zielvariable (*Kriterium*) durch die Werte anderer Variablen (*Prädiktoren, Kovariaten, Kovariablen*) vorherzusagen. Für die statistischen Grundlagen dieser Themen vgl. die darauf spezialisierte Literatur (Eid et al., 2013), die auch für eine vertiefte Behandlung von Regressionsanalysen in R verfügbar ist (Faraway, 2014; Fox & Weisberg, 2011).

6.1 Test auf Korrelation

Die empirische Korrelation zweier normalverteilter Variablen lässt sich daraufhin testen, ob sie mit der H_0 verträglich ist, dass die theoretische Korrelation gleich 0 ist.¹

```
> cor.test(x=<Vektor1>, y=<Vektor2>,
+           alternative=c("two.sided", "less", "greater"), use)
```

Die Daten beider Variablen sind als Vektoren derselben Länge über die Argumente `x` und `y` anzugeben. Alternativ zu `x` und `y` kann auch eine Modellformel `~ <Vektor1> + <Vektor2>` ohne Variablen links der `~` angegeben werden. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Ob die H_1 zwei- ("two.sided"), links- (negativer Zusammenhang, "less") oder rechtsseitig (positiver Zusammenhang, "greater") ist, legt das Argument `alternative` fest. Über das Argument `use` können verschiedene Strategien zur Behandlung fehlender Werte ausgewählt werden (vgl. Abschn. 2.11.4).

```
> N    <- 20                                # Anzahl Beobachtungsobjekte
> DV1 <- rnorm(N, 100, 15)                   # Daten Variable 1
> DV2 <- DV1 + rnorm(N, 0, 50)               # Daten Variable 2
> cor.test(~ DV1 + DV2)                      # oder: cor.test(DV1, DV2)
Pearson's product-moment correlation
data: DV1 and DV2
t = 4.2167, df = 18, p-value = 0.0005186
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
0.3813889 0.8746200
sample estimates:
cor
```

¹Für Tests auf Zusammenhang von ordinalen Variablen vgl. Abschn. 10.3.1.

0.7049359

Die Ausgabe beinhaltet den empirischen t -Wert der Teststatistik (t) samt Freiheitsgraden (df) und zugehörigem p -Wert ($p\text{-value}$) sowie das je nach H_1 ein- oder zweiseitige Vertrauensintervall für die Korrelation, deren empirischer Wert ebenfalls aufgeführt ist. Das Ergebnis lässt sich manuell prüfen:²

```
> r      <- cor(DV1, DV2)                      # empirische Korrelation
> (tVal <- sqrt(N-2) * r / sqrt(1-r^2))    # Teststatistik t-Wert
[1] 4.216709

> (pVal <- 2*pt(tVal, N-2, lower.tail=FALSE))    # p-Wert
[1] 0.00051861

# 95%-Vertrauensintervall für die wahre Korrelation
> fishZ <- 0.5 * log((1+r) / (1-r))          # Fisher Z-Transformation
> fishV <- 1 / (N-3)                          # Varianz der Transformierten
> Zcrit <- qnorm(0.05/2, 0, 1, lower.tail=FALSE) # kritischer Z-Wert
> Zlo   <- fishZ - Zcrit*sqrt(fishV)          # VI für Z untere Grenze
> Zup   <- fishZ + Zcrit*sqrt(fishV)          # VI für Z obere Grenze

# Rücktransformation der Intervallgrenzen
> (ciLo <- tanh(Zlo))
[1] 0.3813889

> (ciUp <- tanh(Zup))
[1] 0.87462
```

Mit `r.test()` aus dem Paket `psych` lassen sich auch Hypothesen darüber testen, ob zwei theoretische Korrelationskoeffizienten aus unabhängigen oder abhängigen Stichproben identisch sind. `rcorr()` aus dem `Hmisc` Paket berechnet für mehrere Variablen die Korrelationsmatrix nach Pearson sowie nach Spearman und testet die resultierenden Korrelationen gleichzeitig auf Signifikanz.

6.2 Einfache lineare Regression

Bei der einfachen linearen Regression werden anhand der paarweise vorhandenen Daten zweier Variablen X und Y die Parameter a und b der Vorhersagegleichung $\hat{Y} = bX + a$ so bestimmt, dass die Werte von Y (dem Kriterium) bestmöglich mit der Vorhersage \hat{Y} aus den Werten von X (dem Prädiktor) übereinstimmen. Dafür muss Y eine quantitative Variable sein, für X sind quantitative und dichotome Variablen möglich. Als Maß für die Güte der Vorhersage wird die Summe der quadrierten Residuen $E = Y - \hat{Y}$, also der Abweichungen von vorhergesagten und Kriteriumswerten herangezogen.³

²Für Fishers Z-Transformation vgl. `FisherZ()`, für die Rücktransformation `FisherZInv()` aus dem Paket `DescTools`.

³Für Maximum-Likelihood-Schätzungen der Parameter vgl. die `glm()` Funktion, deren Anwendung Kap. 8 demonstriert. Eine formalere Behandlung des allgemeinen linearen Modells findet sich in Abschn. 12.9. Für

6.2.1 Deskriptive Modellanpassung

Lineare Modelle wie das der Regression lassen sich mit der `lm()` Funktion anpassen: Sie schätzt die Parameter a und b , zudem ist ihr Ergebnis Grundlage für die Ausgabe der Vorhersage \hat{Y} .

```
> lm(formula=<Modellformel>, data=<Datensatz>, subset=<Indexvektor>,
+     na.action=<Behandlung fehlender Werte>)
```

Unter `formula` ist eine Modellformel der Form `<Kriterium> ~ <Prädiktor>` als Spezifikation des Regressionsmodells anzugeben (vgl. Abschn. 5.2). Soll das Modell ohne den Parameter a gebildet werden, ist ihr `-1` anzuhängen. Stammen die angegebenen Variablen aus einem Datensatz, muss dieser unter `data` übergeben werden. Das Argument `subset` erlaubt es, nur eine Teilmenge der Fälle in die Berechnung einfließen zu lassen, es erwartet einen entsprechenden Indexvektor, der sich auf die Zeilen des Datensatzes bezieht. Mit dem Argument `na.action` kann bestimmt werden, wie mit fehlenden Werten umzugehen ist (vgl. Abschn. 2.11.6).⁴

Als Beispiel soll das Körpergewicht als Kriterium mit der Körpergröße als Prädiktor vorhergesagt werden. Das Körpergewicht wird hier entsprechend einem (sicher unrealistischen) linearen Modell aus der Körpergröße und einem zufälligen Fehler simuliert.

```
> N      <- 100                      # Anzahl Personen
> height <- rnorm(N, 175, 7)          # Prädiktor
> weight <- 0.4*height + 10 + rnorm(N, 0, 3)  # Kriterium
> (fit   <- lm(weight ~ height))       # Regression
Call:
lm(formula = weight ~ height)
```

```
Coefficients:
(Intercept)  height
    27.3351   0.3022
```

Im Ergebnis von `lm()` wird unter der Überschrift `Coefficients` in der Spalte `(Intercept)` die Schätzung für den Schnittpunkt a mit der y -Achse und unter dem Namen des Prädiktors (hier: `height`) die geschätzte Steigung b ausgegeben, die auch als b -Gewicht bezeichnet wird. Das Ergebnis kann manuell verifiziert werden:

```
> (b <- cov(weight, height) / var(height))    # b-Gewicht
[1] 0.3022421

> (a <- mean(weight) - b*mean(height))        # y-Achsenabschnitt
[1] 27.33506

> Yhat <- b*height + a                         # vorhergesagte Werte
```

⁴Methoden zur Einschätzung des Vorhersagefehlers in externen Stichproben vgl. Kap. 13.

⁴In der Voreinstellung `na.omit` zum Ausschluss aller Fälle mit mindestens einem fehlenden Wert ist zu beachten, dass das Ergebnis entsprechend weniger vorhergesagte Werte und Residuen umfasst. Dies kann etwa dann relevant sein, wenn diese Werte mit den ursprünglichen Datenvektoren in einer Rechnung auftauchen und lässt sich vermeiden, indem das Argument auf `na.exclude` gesetzt wird.

Soll statt des b -Gewichts das standardisierte b^z -Gewicht berechnet werden, sind die Variablen in der Modellformel zu z -standardisieren.⁵ Der y -Achsenabschnitt sollte in diesem Fall 0 ergeben, was gerundet der Fall ist. Das b^z -Gewicht ist in der einfachen linearen Regression gleich der Korrelation von Prädiktor und Kriterium.

```
> lm(scale(weight) ~ scale(height))
Call:
lm(formula = scale(weight) ~ scale(height))
Coefficients:
(Intercept)  scale(height)
2.027e-15      5.599e-01

> b * sd(height) / sd(weight)          # Kontrolle
[1] 0.5599206

> cor(height, weight)                # Kontrolle
[1] 0.5599206
```

Ein von `lm()` zurückgegebenes Objekt stellt ein deskriptives Modell der Daten dar, das in anderen Funktionen weiter verwendet werden kann. Es speichert die zur Modellanpassung berechneten Größen als Komponenten einer Liste. Zum Extrahieren der gespeicherten Kennwerte dienen Funktionen wie `residuals()` zum Anzeigen der Residuen, `coef()` zur Ausgabe der Modellparameter, `fitted()` für die vorhergesagten Werte, weiterhin `vcov()` für die Kovarianzmatrix der geschätzten Parameter. Zur Extraktion der Designmatrix i. S. des allgemeinen linearen Modells dient `model.matrix()` (vgl. Abschn. 12.9). Analog liefert `model.frame()` den zur Modellanpassung verwendeten Datensatz zurück. Alle genannten Funktionen erwarten als Argument ein von `lm()` erzeugtes Objekt.

Für eine grafische Veranschaulichung der Regression können die Daten zunächst als Streudiagramm angezeigt werden (Abb. 6.1, vgl. Abschn. 14.2). Die aus der Modellanpassung hervorgehende Schätzung der Regressionsparameter wird dieser Grafik durch `abline(lm-Modell)` in Form eines Geradenabschnitts hinzugefügt (vgl. Abschn. 14.5). Ebenfalls abgebildet wird hier die Gerade der zur Simulation verwendeten fehlerbereinigten Modellgleichung `weight = -0.4*height + 10` und das Zentroid der Daten.

```
# Daten als Streudiagramm darstellen
> plot(weight ~ height, xlab="height [cm]", ylab="weight [kg]", pch=20,
+       main="Daten mit Zentroid, Regressionsgerade und Modellgerade")

> abline(fit, col="blue", lwd=2)          # Regressionsgerade
> abline(a=10, b=0.4, col="gray", lwd=2)    # Modellgerade
> points(mean(weight) ~ mean(height), col="red", pch=4, cex=1.5, lwd=3)
> legend(x="topleft", legend=c("Daten", "Zentroid", "Regressionsgerade",
+      "Modellgerade"), col=c("black", "red", "blue", "gray"),
+      lwd=c(1, 3, 2, 2), pch=c(20, 4, NA, NA), lty=c(NA, NA, 1, 1))
```

⁵Bei fehlenden Werten ist darauf zu achten, dass die z -Standardisierung bei beiden Variablen auf denselben Beobachtungsobjekten beruht. Gegebenenfalls sollten fehlende Werte der beteiligten Variablen aus dem Datensatz vorher manuell ausgeschlossen werden (vgl. Abschn. 2.11.6).

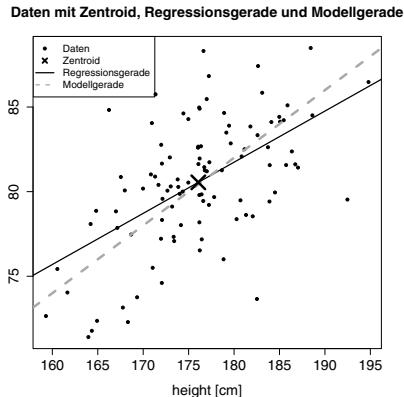


Abbildung 6.1: Lineare Regression: Darstellung der Daten mit Modell- und Regressionsgerade

6.2.2 Regressionsanalyse

Um weitere Informationen und insbesondere inferenzstatistische Kennwerte eines von `lm()` erstellten Modells i. S. einer Regressionsanalyse zu erhalten, wird `summary(⟨lm-Modell⟩)` verwendet.⁶

```
> (sumRes <- summary(fit))
Call:
lm(formula = weight ~ height)

Residuals:
    Min      1Q  Median      3Q     Max 
-8.8433 -2.1565  0.3454  1.8255  7.5915 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 27.33506   7.96062   3.434  0.000874 ***
height       0.30224   0.04518   6.690  1.39e-09 ***  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.143 on 98 degrees of freedom
Multiple R-squared:  0.3135, Adjusted R-squared:  0.3065 
F-statistic: 44.76 on 1 and 98 DF,  p-value: 1.390e-09
```

Die Ausgabe enthält unter der Überschrift **Residuals** eine Zusammenfassung der beobachtungsweisen Residuen. Unter **Coefficients** werden die Koeffizienten (Spalte **Estimate**), ihr

⁶Für eine Mediationsanalyse mit dem Sobel-Test vgl. `sobel()` aus dem `multilevel` Paket (Bliese, 2013). Weitergehende Mediationsanalysen sind mit dem Paket `mediation` (Tingley, Yamamoto, Keele & Imai, 2014) möglich.

Standardfehler (Std. Error), t -Wert (`t value`) und der zugehörige p -Wert für den zweiseitigen t -Test (`Pr(>|t|)`) ausgegeben. Dieser Test wird mit der H_0 durchgeführt, dass das theoretische β -Gewicht gleich 0 ist (vgl. Abschn. 12.9.5). Die Größe des p -Wertes wird mit Sternchen hinter den Werten codiert, deren Bedeutung `Signif. codes` beschreibt.⁷ Die Tabelle der Koeffizienten lässt sich mit `coef()` extrahieren.

```
# geschätzte Koeffizienten, Standardfehler, t-Werte und p-Werte
> coef(sumRes)
   Estimate Std. Error t value Pr(>|t|)
(Intercept) 27.3350625 7.96061681 3.433787 8.737358e-04
height       0.3022421 0.04517855 6.689947 1.390238e-09

# Streuung der Schätzungen: Wurzel aus Diagonale der Kovarianzmatrix
> (sdCoef <- sqrt(diag(vcov(fit))))
(Intercept)      height
7.96061681    0.04517855

> (tVals <- coef(fit) / sdCoef)           # t-Werte
(Intercept)      height
3.433787     6.689947
```

Residual standard error gibt den Standardschätzfehler als Maß für die Diskrepanz zwischen empirischen Werten des Kriteriums und der Modellvorhersage aus. Er ist gleich der Wurzel aus der mittleren Quadratsumme der Residuen als Schätzung der Fehlervarianz, also aus dem Quotienten der Quadratsumme der Residuen und ihrer Freiheitsgrade.

```
# Freiheitsgrade der Residual-Quadratsumme, vgl. fit$df.residual
> P <- 1                               # Anzahl Prädiktoren
> (dfSSE <- N - (P+1))                 # Freiheitsgrade
[1] 98

> SSE <- sum(residuals(fit)^2)          # QS Residuen
> MSE <- SSE / dfSSE                   # mittlere QS Residuen
> sqrt(MSE)                           # Standardschätzfehler
[1] 3.143246
```

Ein weiteres Maß für die Güte der Schätzung ist der Determinationskoeffizient R^2 , in Modellen mit absolutem Term gleich der quadrierten Korrelation zwischen Vorhersage und Kriterium (`Multiple R-squared`, auch multiple Korrelation zwischen Prädiktoren und Kriterium genannt). Das nach Wherry korrigierte R^2 (`Adjusted R-squared`) ist eine lineare Transformation von R^2 und stimmt mit ihm überein, wenn R^2 gleich 1 ist. Im Gegensatz zum empirischen kann das korrigierte R^2 auch negativ werden.

```
# unkorrigiertes  $R^2$ : quadrierte Korrelation Vorhersage mit Kriterium
> Yhat <- fitted(fit)                  # Vorhersage
> (rSq <- cor(Yhat, weight)^2)        # unkorrigiertes  $R^2$ 
[1] 0.3135111
```

⁷Im folgenden wird dieser Teil der Ausgabe mit `options(show.signif.stars=FALSE)` unterdrückt.

```
> 1 - ((N-1) / (N-P-1)) * (1-rSq)          # korrigiertes R^2
[1] 0.3065061
```

Schließlich wird mit einem F -Test für das gesamte Modell die H_0 geprüft, dass (bei einer multiplen Regression, vgl. Abschn. 6.3.1) alle theoretischen β_j -Gewichte gleich 0 sind (vgl. Abschn. 12.9.6, 12.9.7). Die Einzelheiten des Tests sind der empirische Wert des F -Bruchs (F -statistic) gefolgt von den Freiheitsgraden (DF) der Vorhersage und der Residuen sowie dem p -Wert (p-value).

```
> MSpred <- sum((Yhat - mean(Yhat))^2) / P      # mittl. QS Vorhersage
> (Fval <- MSpred / MSE)                         # Teststatistik F-Wert
[1] 44.75539

(pVal <- pf(Fval, P, dfSSE, lower.tail=FALSE))    # p-Wert
[1] 1.390238e-09
```

Für die geschätzten Parameter errechnet `confint(1m-Modell, level=0.95)` das Konfidenzintervall mit der für das Argument `level` angegebenen Breite.

```
> confint(fit)
              2.5 %     97.5 %
(Intercept) 11.5374776 43.1326475
height       0.2125868  0.3918975
```

Die Informationskriterien nach Akaike und Bayes (BIC) berücksichtigen einerseits die Güte der Modellpassung i. S. ihrer maximierten logarithmierten likelihood und bestrafen andererseits die Komplexität des Modells, gemessen an der Anzahl zu schätzender Parameter.⁸ Kleinere Werte stehen für eine höhere Informativität. Bei einer linearen Regression ergibt sich der AIC-Wert direkt aus der Quadratsumme der Residuen sowie der Anzahl zu schätzender Parameter: Bei $p+1$ Koeffizienten für p Prädiktoren und den y -Achsenabschnitt sind dies $p + 1 + 1$.⁹ Besitzen zwei Modelle dieselbe Anpassungsgüte, erhält das Modell mit einer geringeren Anzahl von Parametern den kleineren AIC- bzw. BIC-Wert. Beide Werte werden durch `extractAIC(1m-Modell)` für ein Modell berechnet, wobei für BIC das Argument `k=log(Stichprobengröße)` zu setzen ist.¹⁰ `AIC(1m-Modell)` berechnet den AIC-Wert mit einer anders gewählten Konstante, was die für Modellvergleiche wesentliche Differenz zweier AIC-Werte aber nicht beeinflusst (vgl. Abschn. 6.3.3). Die Ausgabe führt als erstes Element die Anzahl zu schätzender Parameter auf.

```
> extractAIC(fit)                      # AIC-Wert
[1] 2.0000 231.0309

> extractAIC(fit, k=log(N))           # BIC-Wert
[1] 2.0000 236.2413
```

⁸ AIC und BIC besitzen einen engen Bezug zu bestimmten Methoden der Kreuzvalidierung (vgl. Abschn. 13.1).

⁹ Zusätzlich zu β_0 und den β_j ist auch die Fehlerstreuung σ zu schätzen.

¹⁰ Der korrigierte AICc Wert für kleine Stichproben ist mit `aictab()` aus dem Paket `AICcmodavg` (Mazerolle, 2014) berechenbar.

```
> N * log(SSE / N) + 2*(1+1)          # Kontrolle: AIC ...
> AIC(fit)                          # AIC-Berechnung: andere Konstante
[1] 787.1703

> N * (log(2*pi) + log(SSE / N) + 1) + 2*(1+1+1)      # Kontrolle ...
```

6.3 Multiple lineare Regression

Bei der multiplen linearen Regression dienen mehrere quantitative oder dichotome Variablen X_j als Prädiktoren zur Vorhersage des quantitativen Kriteriums Y .¹¹ Die Vorhersagegleichung hat hier die Form $\hat{Y} = b_0 + b_1X_1 + \dots + b_jX_j + \dots + b_pX_p$, wobei die Parameter b_0 und b_j auf Basis der empirischen Daten zu schätzen sind. Dies geschieht wie im Fall der einfachen linearen Regression mit `lm(Modelformel)` nach dem Kriterium der kleinsten Summe der quadrierten Abweichungen von Vorhersage und Kriterium. Die Modellformel hat nun die Form `<Kriterium> ~ <Prädiktor 1> + ... + <Prädiktor p>`, d.h. alle p Prädiktoren X_j werden mit `+` verbunden auf die Rechte Seite der `~` geschrieben.

6.3.1 Deskriptive Modellanpassung und Regressionsanalyse

Als Beispiel soll nun `weight` wie bisher aus der Körpergröße `height` vorhergesagt werden, aber auch mit Hilfe des Alters `age` und später ebenfalls mit der Anzahl der Minuten, die pro Woche Sport getrieben wird (`sport`). Der Simulation der Daten wird die Gültigkeit eines entsprechenden linearen Modells mit zufälligen Fehlern zugrundegelegt.

```
> N      <- 100                      # Anzahl Personen
> height <- rnorm(N, 175, 7)        # Prädiktor 1
> age    <- rnorm(N, 30, 8)         # Prädiktor 2
> sport   <- abs(rnorm(N, 60, 30)) # Prädiktor 3

# Simulation des Kriteriums im Modell der multiplen linearen Regression
> weight <- 0.5*height - 0.3*age - 0.4*sport + 10 + rnorm(N, 0, 3)
> (fitHA <- lm(weight ~ height + age))
Call:
lm(formula = weight ~ height + age)

Coefficients:
(Intercept)  height       age
-72.5404    0.8602   -0.4429
```

Die Schätzungen b_0 und b_j der theoretischen Parameter β_0 und β_j werden in der Ausgabe unter der Überschrift **Coefficients** genannt, wobei b_0 in der Spalte **(Intercept)** und die b_j -Gewichte unter dem Namen des zugehörigen Prädiktors stehen. Auch hier müssen die Variablen

¹¹Für die multivariate multiple Regression mit mehreren Kriteriumsvariablen Y_k vgl. Abschn. 12.5. Eine formalere Behandlung des allgemeinen linearen Modells findet sich in Abschn. 12.9.

in der Modellformel z -standardisiert werden, um die standardisierten b_j^z -Gewichte zu berechnen (vgl. Abschn. 6.2.1, Fußnote 5 für das Vorgehen bei fehlenden Werten). Wie im Fall einer einfachen linearen Regression werden die Parameter mit `summary(⟨lm-Modell⟩)` auf Signifikanz getestet (vgl. Abschn. 6.2.2).

```
> lm(scale(weight) ~ scale(height) + scale(age)) # für stand. b-Gewichte ...
> summary(fitHA) # Regressionsanalyse ...
```

Ist \mathbf{X} die mit `model.matrix(⟨lm-Modell⟩)` erzeugte Designmatrix i. S. des allgemeinen linearen Modells und \mathbf{y} der Vektor der Kriteriumswerte, lassen sich die Schätzungen b_0 und b_j als $\mathbf{X}^+ \mathbf{y}$ berechnen (mit der Pseudoinversen \mathbf{X}^+ von \mathbf{X}).¹² Der Vektor der vorhergesagten Werte berechnet sich als $\hat{\mathbf{y}} = \mathbf{H}\mathbf{y}$ (mit der Hat-Matrix $\mathbf{H} = \mathbf{X}\mathbf{X}^+$, vgl. Abschn. 12.1.7, 12.9.1, 12.9.4).

```
# Koeffizienten aus Produkt der Pseudoinversen X+ mit Kriterium
> X      <- model.matrix(fitHA)           # Designmatrix
> Xplus <- solve(t(X) %*% X) %*% t(X)   # Pseudoinverse X+
> (b      <- Xplus %*% weight)          # Parameterschätzung
(Intercept) -72.5404499
height       0.8602345
age         -0.4428933

# Vorhersage aus Produkt der Hat-Matrix mit Kriterium
> H      <- X %*% Xplus                # Hat-Matrix
> Yhat <- H %*% weight                 # Vorhersage
> all.equal(fitted(fitHA), c(Yhat), check.attributes=FALSE)
[1] TRUE

# bz-Gewichte bei standardisierten Variablen
> S <- cov(cbind(height, age))        # Kovarianzmatrix Prädiktoren
> (1/sd(weight)) * diag(sqrt(diag(S))) %*% b[-1] # Kontrolle ...
```

Die grafische Veranschaulichung mit `scatter3d()` aus dem Paket `car` zeigt die simulierten Daten zusammen mit der Vorhersageebene sowie die Residuen als vertikale Abstände zwischen ihr und den Daten (Abb. 6.2). Als Argument ist dieselbe Modellformel wie für `lm()` zu verwenden. Weitere Argumente kontrollieren das Aussehen der Diagrammelemente. Die Beobachterperspektive dieser Grafik lässt sich interaktiv durch Klicken und Ziehen mit der Maus ändern (vgl. Abschn. 14.8.2).

```
> library(car)                         # für scatter3d()
> scatter3d(weight ~ height + age, fill=FALSE)
```

¹²Es sei vorausgesetzt, dass \mathbf{X} vollen Spaltenrang hat, also keine linearen Abhängigkeiten zwischen den Prädiktoren vorliegen. Dann gilt $\mathbf{X}^+ = (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{X}^t$. Der hier gewählte Rechenweg ist numerisch nicht stabil und weicht von in R-Funktionen implementierten Rechnungen ab (Bates, 2004).

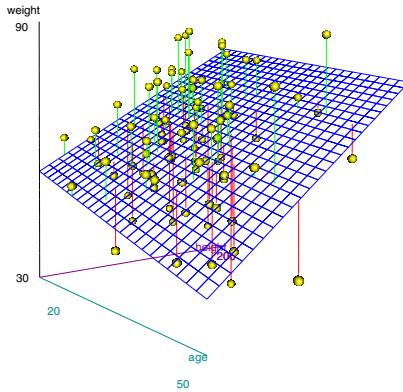


Abbildung 6.2: Daten, Vorhersageebene und Residuen einer multiplen linearen Regression

6.3.2 Modell verändern

Möchte man ein bereits berechnetes Modell nachträglich verändern, ihm etwa einen weiteren Prädiktor hinzufügen, kann dies mit `update()` geschehen. Bei sehr umfassenden Modellen kann dies numerisch effizienter sein, als ein Modell vollständig neu berechnen zu lassen.

```
> update(<lm-Modell>, . ~ . + <weiterer Prädiktor>)
```

Als erstes Argument wird das zu erweiternde Modell eingetragen. Die folgende Modellformel `. ~ .` signalisiert, dass alle bisherigen Prädiktoren beizubehalten sind. Jeder weitere Prädiktor wird dann mit einem `+` angefügt. Das Ergebnis von `update()` ist das aktualisierte Modell.

```
# füge sport als zusätzlichen Prädiktor hinzu
> (fitHAS <- update(fitHA, . ~ . + sport))
Call:
lm(formula = weight ~ height + age + sport)
```

Coefficients:

	(Intercept)	height	age	sport
	8.1012	0.5149	-0.2866	-0.4160

Auf die gleiche Weise wie Prädiktoren hinzugefügt werden können, lassen sie sich auch entfernen, wobei statt des `+` ein `-` zu verwenden ist.

```
# entferne Prädiktor height
> (fitAS <- update(fitHAS, . ~ . - height))
Call:
lm(formula = weight ~ age + sport)
```

Coefficients:

	(Intercept)	age	sport
	98.5145	-0.2459	-0.4452

```
> (fitH <- update(fitHA, . ~ . - age)) # entferne age ...
```

6.3.3 Modelle vergleichen und auswählen

Existieren bei einer multiplen Regression viele potentielle Prädiktoren, stellt sich die Frage, welche letztlich im Modell berücksichtigt werden sollten. In einem hierarchischen Vorgehen lässt sich dafür testen, inwieweit die Hinzunahme von Prädiktoren zu einer bedeutsamen Verbesserung der Modellpassung führt. Dies ist in Form von *F*-Tests über Modellvergleiche möglich, die die Veränderung der Residualvarianz in Abhängigkeit vom verwendeten Prädiktorensatz testen. Dabei lassen sich sinnvoll nur *nested* Modelle mit demselben Kriterium vergleichen, bei denen der Prädiktorensatz eines eingeschränkten Modells vollständig im Prädiktorensatz des umfassenderen Modells enthalten ist, das zusätzlich noch weitere Prädiktoren berücksichtigt (vgl. Abschn. 12.9.6).

Um verschiedene Regressionsmodelle miteinander hinsichtlich der Quadratsumme der Residuen sowie Akaikes Informationskriterium AIC vergleichen zu können, lassen sich folgende Funktionen verwenden:

```
> step(object=lm-Modell, scope=~ . + <Prädiktoren>)
> add1(object=lm-Modell, scope=~ . + <Prädiktoren>, test="none")
> drop1(object=lm-Modell, scope=~ . - <Prädiktoren>, test="none")
```

`step()` prüft im *Stepwise*-Verfahren sequentiell Teilmengen von denjenigen Prädiktoren der unter `object` angegebenen Regression, die in `scope` als Modellformel definiert werden. Diese Modellformel beginnt mit einer `~` und enthält danach alle Prädiktoren aus `object` (Voreinstellung `~ .`) sowie ggf. weitere mit `+` hinzuzufügende. `step()` berechnet in jedem Schritt die Auswirkung einer Modellveränderung durch Hinzunahme oder Weglassen eines Prädiktors auf Vergleichskriterien und wählt schließlich ein Modell, das durch diese Schritte nicht substantiell verbessert werden kann.¹³

`add1()` berechnet separat den Effekt der Hinzunahme jeweils eines unter `scope` genannten Prädiktors zu einer unter `object` angegebenen Regression. Die Änderung in der Quadratsumme der Residuen bei jedem Schritt kann inferenzstatistisch geprüft werden, indem das Argument `test="F"` gesetzt wird. Das einfachste Modell, in dem kein Prädiktor berücksichtigt wird, lautet `<Kriterium> ~ 1` und sorgt dafür, dass für jeden Wert der konstante Mittelwert des Kriteriums vorhergesagt wird.

```
# Vergleich: nur height als Prädiktor, sowie auch age bzw. sport
> add1(fitH, . ~ . + age + sport, test="F")
Single term additions
Model:
weight ~ height
Df  Sum of Sq      RSS      AIC  F value    Pr(>F)
```

¹³Das Paket `leaps` (Lumley, 2009) ermöglicht die automatisierte Auswahl aller Teilmengen von Prädiktoren. Beide Verfahren sind mit vielen inhaltlichen Problemen verbunden, für eine Diskussion und verschiedene Strategien zur Auswahl von Prädiktoren vgl. Miller (2002). Für penalisierte Regressionsverfahren, die auch eine Auswahl von Prädiktoren vornehmen, vgl. Abschn. 6.6.1.

```
<none>           14457.9  501.38
age      1    1246.3  13211.6  494.37     9.15   0.003183 ***
sport     1    12962.8  1495.1  276.48    840.98 < 2.2e-16 ***
```

In der Zeile `<none>` der Ausgabe finden sich die Kennwerte des eingeschränkten `object` Modells. Die folgenden Zeilen nennen die Kennwerte der umfassenderen Modelle, bei denen zusätzlich jeweils einer der unter `scope` genannten Prädiktoren berücksichtigt wird. Die Spalte `Sum of Sq` führt die sequentielle Quadratsumme vom Typ I des zusätzlichen Prädiktors beim Wechsel hin zu einem solchen Modell auf. Sie ist gleich der Differenz der in der Spalte `RSS` genannten Quadratsummen der Residuen (*residual sum of squares*) und damit ein Maß für die Reduktion an Fehlervarianz beim Modellwechsel. Die inferenzstatistischen Kennwerte finden sich in den Spalten `Df` für die Differenz der Fehler-Freiheitsgrade vom eingeschränkten und umfassenderen Modell, `F value` für den *F*-Wert und `Pr(>F)` für den *p*-Wert. Auf ähnliche Weise kann mit `drop1()` die Quadratsumme vom Typ III eines Vorhersageterms als Effekt des Weglassens jeweils eines Prädiktors berechnet werden (vgl. Abschn. 7.5.2).

Der *F*-Wert ergibt sich als Quotient der angegebenen Quadratsumme geteilt durch die Differenz der Fehler-Freiheitsgrade und der Residual-Quadratsumme des umfassenderen Modells geteilt durch ihre Freiheitsgrade (vgl. Abschn. 12.9.7). Die Fehler-Freiheitsgrade eines Modells berechnen sich als Differenz zwischen der Stichprobengröße und der Anzahl zu schätzender Parameter. Hier sind dies im eingeschränkten Modell mit einem Regressionsgewicht und dem absoluten Term zwei, in jedem Modell mit einem zusätzlichen Prädiktor entsprechend drei.

```
# RSS Zeile <none>: Prädiktor height
> (rssH <- sum(residuals(lm(weight ~ height))^2))
[1] 14457.9

# RSS Zeile age: Prädiktoren height und age
> (rssHA <- sum(residuals(lm(weight ~ height + age))^2))
[1] 13211.64

# RSS Zeile sport: Prädiktoren height und sport
> (rssHS <- sum(residuals(lm(weight ~ height + sport))^2))
[1] 1495.140

# Fehler-Freiheitsgrade der drei Modelle
> dfEH  <- N - (1+1)                      # eingeschränktes Modell
> dfEHA <- dfEHS <- N - (2+1)            # beide umfassenderen Modelle

# mittlere partielle Effekt-QS für Hinzunahme von age
> MSha <- (rssH - rssHA) / (dfEH - dfEHA)

# mittlere Residual-QS umfassenderes Modell: Prädiktoren height und age
> MSEha <- rssHA / dfEHA

# F-Wert für Effekt der Hinzunahme von age
> (Fha <- MSha / MSEha)
[1] 9.150025
```

```
# analog für Hinzunahme von sport
> MShs <- (rssH - rssHS) / (dfEH - dfEHS)
> MSEhs <- rssHS / dfEHS
> (Fhs <- MShs / MSEhs)
[1] 840.9835
```

Die Spalte AIC listet den Wert von Akaiakes Informationskriterium für beide Modelle auf.

Um nested Modelle gegeneinander zu testen, die sich um mehr als einen Prädiktor unterscheiden, können mit `lm()` erstellte Regressionsen an die `anova()` Funktion übergeben werden, die sich auch für Varianzanalysen eignet (vgl. Abschn. 7.3.3). Die Modelle müssen sich auf dieselben Beobachtungen beziehen – bei fehlenden Werten ist sicherzustellen, dass in beide Modelle dieselben Beobachtungen einfließen.

```
> anova(<eingeschränktes lm-Modell>, <umfassenderes lm-Modell>)
```

Im Beispiel soll das Kriterium `weight` entweder nur durch `height`, oder aber durch alle drei Prädiktoren vorhergesagt werden.

```
> anova(fitH, fitHAS)      # F-Test für Hinzunahme von age und sport
Analysis of Variance Table
Model 1: weight ~ height
Model 2: weight ~ height + age + sport
  Res.Df   RSS Df Sum of Sq    F    Pr(>F)
1     98 14457.9
2     96  979.6  2    13478 660.4 < 2.2e-16 ***
```

In der Ausgabe wird in der Spalte `RSS` die Residual-Quadratsumme für jedes Modell aufgeführt, die zugehörigen Freiheitsgrade in der Spalte `Res.Df`. In der Spalte `Df` steht deren Differenz, um wie viele Parameter sich die Modelle also unterscheiden. Die Reduktion der Residual-Quadratsumme beim Wechsel zum umfassenderen Modell findet sich in der Spalte `Sum of Sq`, daneben der zugehörige *F*-Wert (`F`) und *p*-Wert (`Pr(>F)`).

6.3.4 Moderierte Regression

Bei der multiplen Regression kann die Hypothese bestehen, dass die theoretischen Regressionsparameter eines Prädiktors von der Ausprägung eines anderen Prädiktors abhängen (Aiken & West, 1991). Im Rahmen eines einfachen kausalen Einflussmodells mit einem Prädiktor (UV), einem Kriterium (AV) und einer Drittvariable ließe sich etwa vermuten, dass der Einfluss der UV auf die AV davon abhängt, wie die Drittvariable ausgeprägt ist.¹⁴ Bei quantitativen Variablen spricht man dann von einer *Moderation*, während man im Rahmen einer Varianzanalyse mit mehreren kategorialen UVn den Begriff der *Interaktion* verwendet (vgl. Abschn. 7.5). Die Vorhersagegleichung der Regression mit zwei Prädiktoren erweitert sich durch den Interaktionsterm zu $\hat{Y} = b_0 + b_1X_1 + b_2X_2 + b_3(X_1X_2)$. Es wird also einfach ein weiterer

¹⁴Für Hinweise zur Analyse komplexerer Kausalmodelle vgl. Abschn. 12.3, Fußnote 32.

Vorhersageterm hinzugefügt, der gleich dem Produkt beider Prädiktoren ist. Im Aufruf von `lm()` geschieht dies durch Aufnahme des Terms `X1:X2` in die Modellformel.

Im umgeformten Modell $\hat{Y} = (b_0 + b_2 X_2) + (b_1 + b_3 X_2) X_1$ wird ersichtlich, dass es sich wie jenes der einfachen linearen Regression des Prädiktors X_1 schreiben lässt, wobei y -Achsenabschnitt und Steigungsparameter nun lineare Funktionen des als Moderator betrachteten Prädiktors X_2 sind. Der Term $b_{0s} = b_0 + b_2 X_2$ wird als *simple intercept*, $b_{1s} = b_1 + b_3 X_2$ als *simple slope* bezeichnet. Für einen festen Wert des Moderators X_2 ergeben sich für b_{0s} und b_{1s} konkrete Werte, die dann *bedingte* Koeffizienten heißen. Um einen Eindruck von der Bandbreite der Parameterschätzungen zu gewinnen, bieten sich für die Wahl fester Werte von X_2 dessen Mittelwert sowie die Werte \pm eine Standardabweichung um den Mittelwert an. Eine andere Möglichkeit sind der Median sowie das erste und dritte Quartil von X_2 .

`plotSlopes()` aus dem Paket `rockchalk` (Johnson, 2014) berechnet die bedingten Koeffizienten für vorgegebene Werte des Moderators – auch für Modelle mit einem Moderator, aber noch weiteren Prädiktoren, die nicht Teil einer Interaktion sind.

```
> plotSlopes(<(lm-Modell>, plotx="<Prädiktor>", modx="<Moderator>",
+             modxVals="<Moderator-Werte>")
```

Als erstes Argument ist ein mit `lm()` erstelltes Regressionsmodell zu übergeben. `plotx` erwartet den Namen von X_1 , also der Variable, die Teil der Interaktion mit dem Moderator ist. Im ausgegebenen Streudiagramm definiert sie die x -Achse. Für `modx` ist X_2 als Moderatorvariable zu nennen. Für welche Werte des Moderators die bedingten Koeffizienten berechnet werden sollen, kontrolliert `modxVals`: Auf "std.dev" gesetzt sind dies der Mittelwert \pm eine Standardabweichung, mit "quantile" der Median sowie das erste und dritte Quartil. Das Ergebnis ist ein Streudiagramm von X_1 und Y , in das die zu b_{0s} und b_{1s} gehörenden bedingten Regressionsgeraden eingezeichnet sind (Abb. 6.3). Moderator sei hier das Alter bei der Regression des Körpergewichts auf die Körpergröße.

```
# Modell mit zentrierten Prädiktoren und Interaktionsterm
> heightC <- c(scale(height, center=TRUE, scale=FALSE))
> ageC     <- c(scale(age,    center=TRUE, scale=FALSE))
> fitHAI   <- lm(weight ~ heightC + ageC + heightC:ageC)
> coef(summary(fitHAI))
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 63.95331    1.17188  54.573  < 2e-16 ***
heightC      0.85591    0.16413   5.215  1.06e-06 ***
ageC        -0.44947    0.14696  -3.058  0.00288 **
heightC:ageC  0.01509    0.01945   0.776  0.43986

# Diagramm mit simple slopes für drei Werte des Moderators
> library(rockchalk)           # für plotSlopes(), testSlopes()
> ps <- plotSlopes(fitHAI, plotx="heightC", modx="ageC",
+                   modxVals="std.dev")
```

Als weiteres Ergebnis liefert `plotSlopes()` ein Objekt, das an `testSlopes()` übergeben werden kann, um Standardabweichungen, t - und p -Werte für die Tests der bedingten Koeffizienten

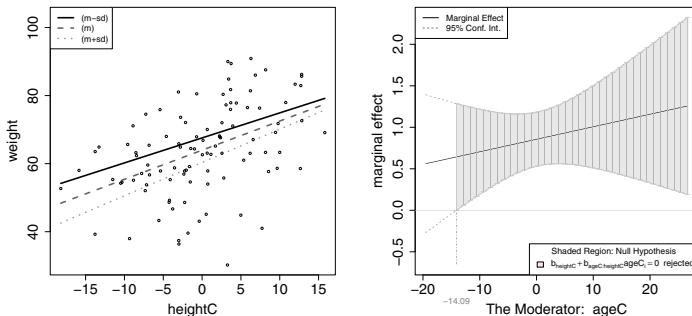


Abbildung 6.3: Simple slopes in der Regression mit Moderator

zu berechnen. In der von `testSlopes()` ausgegebenen Liste stehen diese Ergebnisse in der Komponente `hypotests`.

```
> (ts <- testSlopes(ps))
Values of ageC INSIDE this interval:
      lo          hi
-14.08742  35.21252
cause the slope of (b1 + b2*ageC)heightC to be statistically significant

$hypotests
  "ageC"    slope Std. Error   t value   Pr(>|t|)
(m-sd) -8.03  0.7347739  0.2303663 3.189589 1.924802e-03
(m)     0.00  0.8559139  0.1641263 5.214971 1.055984e-06
(m+sd)  8.03  0.9770540  0.2226857 4.387592 2.941003e-05

...
# gekürzte Ausgabe
```

Die Ausgabe nennt zunächst das Johnson-Neyman Intervall als Wertebereich des Moderators, für den der Test der bedingten Steigung signifikant ist. In den Zeilen der Komponente `hypotests` stehen die Kennwerte der bedingten Steigungen, die sich für den Mittelwert von X_2 ((\bar{m})) und \pm eine Standardabweichung von X_2 ergeben (($\bar{m}-sd$) bzw. ($\bar{m}+sd$)). In der Spalte `slope` findet sich die bedingte Steigung, dahinter die Standardabweichung ihrer Schätzung (`Std. Error`) sowie der t -Wert (`t value`) und der zugehörige zweiseitige p -Wert (`Pr(>|t|)`).

Darüber hinaus lässt sich die von `testSlopes()` erzeugte Liste an `plot()` übergeben, um ein Diagramm zu erstellen, das den Konfidenzbereich um die bedingten Steigungen für den beobachteten Bereich der Werte des Moderators visualisiert (Abb. 6.3).

```
> plot(ts)      # Diagramm für Konfidenzbereich um bedingte Steigung

# manuelle Kontrolle
> coeffs <- coef(fitHAI)           # extrahiere Koeffizienten
> b0 <- coeffs[1]
```

```
> b1 <- coeffs[2]
> b2 <- coeffs[3]
> b3 <- coeffs[4]

# bedingte y-Achsenabschnitte
> b0 + b2*(mean(ageC) - sd(ageC))      # für M - 1sd ...
> b0 + b2* mean(ageC)                   # für M ...
> b0 + b2*(mean(ageC) + sd(ageC))      # für M + 1sd ...

# bedingte Steigungen
> b1 + b3*(mean(ageC) - sd(ageC))      # für M - 1sd ...
> b1 + b3* mean(ageC)                   # für M ...
> b1 + b3*(mean(ageC) + sd(ageC))      # für M + 1sd ...
```

6.4 Regressionsmodelle auf andere Daten anwenden

`predict()` wendet ein von `lm()` angepasstes Regressionsmodell auf neue Daten an, um für sie die Vorhersage \hat{Y} zu berechnen (vgl. `?predict.lm`).

```
> predict(object=<lm-Modell>, newdata=<Datensatz>, se.fit=FALSE,
+         interval=NULL, level=<Breite Konfidenzintervall>)
```

Als erstes Argument ist ein von `lm()` erzeugtes Objekt zu übergeben. Werden alle weiteren Argumente weggelassen, liefert die Funktion die Vorhersage für die ursprünglichen Prädiktorwerte zurück, also `fitted(<Modell>)`. Wird unter `newdata` ein Datensatz übergeben, der Variablen mit denselben Namen wie jene der ursprünglichen Prädiktoren enthält, so wird \hat{Y} für die Werte dieser neuen Variablen berechnet.¹⁵ Sollen die Standardabweichungen für \hat{Y} ermittelt werden, ist `se.fit=TRUE` zu setzen.

Mit dem Argument `interval="confidence"` berechnet `predict()` zusätzlich für jeden (bereits erhobenen) Wert der Prädiktorvariablen die Grenzen des zugehörigen Konfidenzintervalls, dessen Breite mit `level` kontrolliert wird. Mit `interval="prediction"` erhält man die Grenzen des Toleranzintervalls für die Vorhersage auf Basis neuer Daten, die nicht in die Modellanpassung eingeflossen sind. Die Ausgabe erfolgt in Form einer Matrix, deren erste Spalte (`fit`) die Vorhersage ist, während die beiden weiteren Spalten untere (`lwr`) und obere Grenzen (`upr`) des Vertrauensbereichs nennen.

Im Beispiel des Modells mit nur dem Prädiktor `height` wird für den Aufruf von `predict()` zunächst ein Datensatz mit einer passend benannten Variable erzeugt.

```
> newHeight <- c(177, 150, 192, 189, 181)          # neue Daten
> newDf     <- data.frame(height=newHeight)        # Datensatz
> predict(fitH, newDf, interval="prediction", level=0.95)
```

¹⁵Handelt es sich etwa im Rahmen einer Kovarianzanalyse (vgl. Abschn. 7.8) um einen kategorialen Prädiktor – ein Objekt der Klasse `factor`, so muss die zugehörige Variable in `newdata` dieselben Stufen in derselben Reihenfolge beinhalten wie die Variable des ursprünglichen Modells – selbst wenn nicht alle Faktorstufen tatsächlich als Ausprägung vorkommen.

```

fit      lwr      upr
1 66.00053 41.76303 90.23803
2 43.68101 18.07599 69.28604
3 78.40026 53.47502 103.32550
4 75.92032 51.21385 100.62678
5 69.30713 44.98704 93.62721

# Kontrolle der Vorhersage durch Extrahieren der Regressionsparameter
# und Einsetzen der neuen Daten in die Vorhersagegleichung
> (coeffs <- coef(fitH))                                # Parameter
(Intercept)      height
-80.3163014   0.8266488

> coeffs[2]*newHeight + coeffs[1]                         # Vorhersage
[1] 66.00053 43.68101 78.40026 75.92032 69.30713

```

Der Vertrauensbereich um die Vorhersage für die ursprünglichen Daten lässt sich auch grafisch darstellen (Abb. 6.4).

```

> plot(weight ~ height, pch=20, xlab="Prädiktor", ylab="Kriterium und
+       Vorhersage", xaxs="i", main="Daten und Vorhersage durch Regression")

# Vertrauensintervall um Vorhersage für Originaldaten
> predOrg <- predict(fitH, interval="confidence", level=0.95)
> hOrd    <- order(height)
> polygon(c(height[hOrd],           height[rev(hOrd)]),
+           c(predOrg[hOrd, "lwr"], predOrg[rev(hOrd), "upr"]),
+           border=NA, col=rgb(0.7, 0.7, 0.7))
# Regressionsgerade
> abline(fitH, col="blue")                               # Regressionsgerade
> legend(x="bottomright", legend=c("Daten", "Vorhersage",
+         "Vertrauensbereich"), pch=c(20, NA, NA), lty=c(NA, 1, 1),
+         lwd=c(NA, 1, 8), col=c("black", "blue", "gray"))

```

6.5 Regressionsdiagnostik

Einer konventionellen Regressionsanalyse liegen verschiedene Annahmen zugrunde, deren Gültigkeit vorauszusetzen ist, damit die berechneten Standardfehler der Parameterschätzungen und die p -Werte korrekt sind (vgl. Abschn. 12.9.4). Dazu gehören Normalverteiltheit und gemeinsame Unabhängigkeit der Messfehler des Kriteriums, die zudem unabhängig von den Prädiktoren sein müssen (vgl. Abschn. 12.9.4). Hinzu kommt die *Homoskedastizität*, also die Gleichheit aller bedingten Fehlervarianzen (Abb. 12.5).

Mit Hilfe der Regressionsdiagnostik soll zum einen geprüft werden, ob die Daten mit den gemachten Annahmen konsistent sind. Zum anderen kann die Parameterschätzung der konventionellen Regression durch wenige Ausreißer überproportional beeinflusst werden. Daher ist es

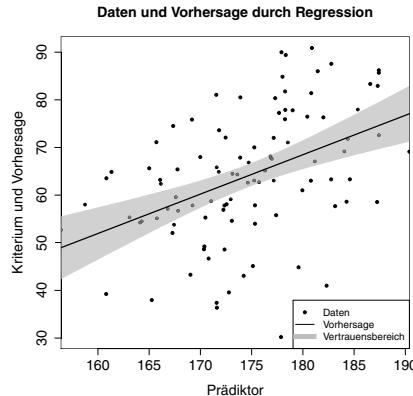


Abbildung 6.4: Lineare Regression: Prädiktor, Kriterium und Vorhersage mit Vertrauensbereich

von Interesse, diese zu identifizieren und den Einfluss einzelner Beobachtungen auf das Ergebnis zu bestimmen. Schließlich ist in der multiplen Regression das Ausmaß der Multikollinearität bedeutsam, also die wechselseitige lineare Abhängigkeit der Prädiktoren untereinander. Für eine ausführliche Darstellung vgl. Fox und Weisberg (2011) sowie das zugehörige Paket **car** für Funktionen, mit denen sich eine Vielzahl diagnostischer Diagramme erstellen lassen.

6.5.1 Extremwerte, Ausreißer und Einfluss

Unter einem Ausreißer sind hier Beobachtungen zu verstehen, deren Kriteriumswert stark vom Kriteriumswert anderer Beobachtungen abweicht, die ähnliche Prädiktorwerte besitzen. Extremwerte einer gegebenen Variable zeichnen sich dadurch aus, dass sie weit außerhalb der Verteilung der übrigen Beobachtungen liegen. Die grafische Beurteilung, ob Extremwerte vorliegen, lässt sich getrennt für jede Variable etwa durch Histogramme oder boxplots der z -standardisierten Variable durchführen (Abb. 6.5; vgl. Abschn. 14.6.1, 14.6.3).

Als numerische Indikatoren eignen sich die z -standardisierten Werte, an denen abzulesen ist, wie viele Streuungseinheiten ein Wert vom Mittelwert entfernt ist. Als numerisches Maß der multivariaten Extremwertanalyse bietet sich die Mahalanobisdistanz als Verallgemeinerung der z -Transformation an (vgl. Abschn. 12.1.4). Sie repräsentiert den Abstand eines Datenpunkts zum Zentroid der Verteilung, wobei der Gesamtform der Verteilung i. S. der Kovarianzmatrix Rechnung getragen wird. Für die grafische Darstellung der gemeinsamen Verteilung von zwei oder drei Variablen vgl. Abschn. 14.6.8, 14.8.¹⁶

```
> Xpred <- cbind(height, age, sport)      # Prädiktoren als Datenmatrix
> Xz <- scale(Xpred)                      # z-Transformierte
```

¹⁶Da Extremwerte die Lage und Streuung der Daten mit beeinflussen, sollten hierfür evtl. robuste Schätzer in Betracht gezogen werden (Rousseeuw & van Zomeren, 1990). Robuste Schätzungen für die Kovarianzmatrix können etwa an das Argument `cov` von `mahalanobis()` übergeben werden (vgl. Abschn. 2.7.9). Für fortgeschrittene Tests, ob Ausreißer in multivariaten Daten vorliegen, vgl. `aq.plot()` und `pcout()` aus dem Paket **mvoutlier** (Filzmoser & Gschwandtner, 2014).

```
> boxplot(Xz, main="Verteilung z-standard. Prädiktoren")
> summary(Xz)
      height           age           sport
Min.   :-2.534e+00  Min.   :-1.952e+00  Min.   :-1.796e+00
1st Qu.:-6.040e-01  1st Qu.:-7.110e-01  1st Qu.:-6.884e-01
Median : 1.075e-02  Median :-1.069e-01  Median :-9.467e-02
Mean   : 1.781e-15  Mean   :-8.978e-17  Mean   : 1.087e-16
3rd Qu.: 6.338e-01  3rd Qu.: 7.411e-01  3rd Qu.: 6.519e-01
Max.   : 2.200e+00  Max.   : 2.839e+00  Max.   : 2.727e+00

# Mahalanobis-Distanz der Beobachtungen zum Zentroid der Prädiktoren
> ctrX <- colMeans(Xpred)                      # Zentroid
> sX    <- cov(Xpred)                         # Kovarianzmatrix
> mahaSq <- mahalanobis(Xpred, ctrX, sX)       # quadrierte M-Distanzen
> summary(sqrt(mahaSq))
      Min. 1st Qu. Median  Mean 3rd Qu.  Max.
0.2855  1.1640  1.5940  1.5970  1.9750  3.3090
```

Durch Extremwerte oder Ausreißer wird die ermittelte Vorhersagegleichung womöglich in dem Sinne verzerrt, dass sie die Mehrzahl der Daten nicht mehr gut repräsentiert. Um den Einfluss der einzelnen Beobachtungen auf die Parameterschätzungen der Regression direkt zu quantifizieren, existieren verschiedene Kennwerte, darunter der Hebelwert h (*leverage*). h wird durch `hatvalues(`(`lm`-Modell)) berechnet.¹⁷ Für Modelle, die einen absoluten Term b_0 einschließen, kann h Werte im Intervall $[\frac{1}{n}, 1]$ annehmen, wobei n die Anzahl an Beobachtungen ist. Der Mittelwert ist dann gleich $\frac{p+1}{n}$ mit $p+1$ als Anzahl zu schätzender Parameter der Regression (p Prädiktoren sowie absoluter Term). Um besonders große Hebelwerte zu identifizieren, kann ihre Verteilung etwa über ein Histogramm oder einen *spike-plot* veranschaulicht werden (Abb. 6.5, vgl. Abschn. 14.2). Als numerisches Kriterium für auffällig große Hebelwerte dient bisweilen das Zwei- bis Dreifache seines Mittelwerts.

```
> fitHAS <- lm(weight ~ height + age + sport)      # Regression
> h       <- hatvalues(fitHAS)                      # Hebelwerte
> hist(h, main="Histogramm der Hebelwerte")        # Histogramm
> summary(h)
      Min. 1st Qu. Median  Mean 3rd Qu.  Max.
0.01082  0.02368  0.03568  0.04000  0.04942  0.12060
```

Zur Kontrolle lässt sich die Beziehung nutzen, dass die quadrierte Mahalanobisdistanz einer Beobachtung i zum Zentroid der Prädiktoren gleich $(n - 1)(h_i - \frac{1}{n})$ ist.

```
> all.equal(mahaSq, (N-1) * (h - (1/N)))
[1] TRUE
```

Die Indizes DfFITS bzw. DfBETAS liefern für jede Beobachtung ein standardisiertes Maß, wie stark sich die Vorhersagewerte (DfFITS) bzw. jeder geschätzte Parameter (DfBETAS) tatsächlich ändern, wenn die Beobachtung aus den Daten ausgeschlossen wird. In welchem Ausmaß sich

¹⁷Zudem ist h_i gleich dem i -ten Eintrag $\overline{\mathbf{H}_{ii}}$ in der Diagonale der Hat-Matrix \mathbf{H} (vgl. Abschn. 6.3.1).

dabei der Standardschätzfehler ändert, wird über das Verhältnis beider resultierenden Werte (mit bzw. ohne ausgeschlossene Beobachtung) ausgedrückt. Cooks Distanz ist ein weiteres Einflussmaß, das sich als $\frac{E^2}{\hat{\sigma}^2(1-h)^2} \cdot \frac{h}{p+1}$ berechnet, wobei E für die Residuen $Y - \hat{Y}$ und $\hat{\sigma}$ für den Standardschätzfehler der Regression mit $p + 1$ Parametern steht.

Mit `influence.measures(<lm-Modell>)` lassen sich die genannten Kennwerte gleichzeitig berechnen. Auffällige Beobachtungen können aus der zurückgegebenen Liste mit `summary()` extrahiert werden, wobei der Wert der abweichenden diagnostischen Größe durch einen Stern * markiert ist.

```
> inflRes <- influence.measures(fitHAS)      # Diagnosegrößen
> summary(inflRes)                          # auffällige Beobachtungen
Potentially influential observations of
lm(formula = weight ~ height + age + sport) :
   dfb.1 dfb.hght dfb.age dfb.spht dffit cov.r cook.d hat
6   -0.05     0.07   -0.23    0.25   0.40  0.86_*  0.04  0.03
13  -0.21     0.21   -0.10    0.36   0.41  1.13_*  0.04  0.12_*
30   0.19    -0.23    0.17   -0.01  -0.42  0.74_*  0.04  0.02
31  -0.08     0.11   -0.08   -0.05   0.27  0.86_*  0.02  0.01
67   0.02    -0.02   -0.03    0.03  -0.05  1.17_*  0.00  0.11
72  -0.03     0.03    0.04    0.01   0.05  1.17_*  0.00  0.11
86   0.10    -0.16    0.60   -0.25   0.68_*  0.90  0.11  0.08
97  -0.01     0.01   -0.01    0.01   0.02  1.16_*  0.00  0.10
```

In der Ausgabe beziehen sich die Spalten `dfb.<Prädiktor>` auf das DfBETA jedes Prädiktors (inkl. des absoluten Terms 1 in der ersten Spalte), `dffit` auf DfFITS, `cov.r` auf das Verhältnis der Standardschätzfehler, `cook.d` auf Cooks Distanz und `hat` auf den Hebelwert. Für Funktionen zur separaten Berechnung der Maße vgl. `?influence.measures`.

```
> cooksDst <- cooks.distance(fitHAS)          # Cooks Distanz
> plot(cooksDst, main="Cooks Distanz", type="h") # spike-plot

# manuelle Berechnung
> P   <- 3                                     # Anzahl Prädiktoren
> E   <- residuals(fitHAS)                     # Residuen
> MSE <- sum(E^2) / (N - (P+1))               # quadr. Standardschätzfehler
> CD  <- (E^2 / (MSE * (1-h)^2)) * (h / (P+1)) # Cooks Distanz
> all.equal(cooksDst, CD)                      # Kontrolle
[1] TRUE
```

Grafisch aufbereitete Informationen über den Einfluss einzelner Beobachtungen sowie über die Verteilung der Residuen (s. u.) liefert auch eine mit `plot(<lm-Modell>, which=1:6)` aufzurufende Serie von Diagrammen. Über das Argument `which` können dabei einzelne Grafiken der Serie selektiv gezeigt werden. Vergleiche dazu auch `influencePlot()` und `influenceIndexPlot()` aus dem `car` Paket.

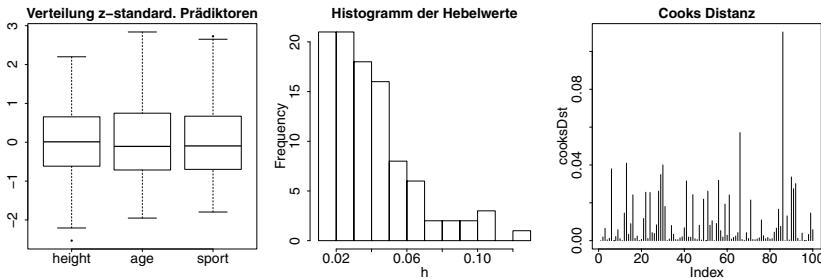


Abbildung 6.5: Beurteilung von Extremwerten und Einflussgrößen in der Regression

6.5.2 Verteilungseigenschaften der Residuen

Anhand verschiedener grafischer Darstellungen der Residuen einer Regression lässt sich heuristisch beurteilen, ob die vorliegenden Daten mit den Voraussetzungen der Normalverteiltheit, Unabhängigkeit und Homoskedastizität der Messfehler vereinbar sind. Als Grundlage können die Residuen $E = Y - \hat{Y}$ selbst, oder aber zwei Transformationen von ihnen dienen: Die *standardisierten* und *studentisierten* Residuen besitzen eine theoretische Streuung von 1 und ergeben sich als $\frac{E}{\hat{\sigma}\sqrt{1-h}}$, wobei $\hat{\sigma}$ eine Schätzung der theoretischen Fehlerstreuung ist (vgl. Abschn. 12.9.4).

Für die standardisierten Residuen wird der Standardschätzfehler als globale Schätzung in der Rolle von $\hat{\sigma}$ verwendet, bei den studentisierten Residuen dagegen eine beobachtungsweise Schätzung $\hat{\sigma}_{(i)}$. Dabei wird im Fall *extern* studentisierter Residuen (*leave-one-out*) $\hat{\sigma}_{(i)}$ für jede Beobachtung i auf Basis des Regressionsmodells berechnet, in das alle Daten bis auf die der i -ten Beobachtung einfließen. Für die im folgenden aufgeführten Prüfmöglichkeiten werden oft standardisierte oder studentisierte Residuen E gegenüber vorgezogen.

`lm.influence()` speichert $\hat{\sigma}_{(i)}$ in der Komponente `sigma` der ausgegebenen Liste. Die Residuen selbst lassen sich mit `residuals()` für E , `rstandard()` für die standardisierten und `rstudent()` für die extern studentisierten Residuen ermitteln. Bei allen Funktionen ist als Argument ein von `lm()` erstelltes Modell zu übergeben.

```
> Estnd <- rstandard(fitHAS) # standardisierte Residuen
> all.equal(Estnd, E / sqrt(MSE * (1-h))) # manuelle Kontrolle
[1] TRUE

# studentisierte Residuen und manuelle Kontrolle
> Estud <- rstudent(fitHAS)
> all.equal(Estud, E / (lm.influence(fitHAS)$sigma * sqrt(1-h)))
[1] TRUE
```

Für eine visuell-exploratorische Beurteilung der Normalverteiltheit wird die Verteilung der bevorzugten Residuen-Variante mit einem Histogramm der relativen Klassenhäufigkeiten dargestellt, dem die Dichtefunktion der Standardnormalverteilung hinzugefügt wurde (Abb. 6.6, vgl. Abschn. 14.6.1). Das Histogramm sollte in seiner Form nicht stark von der Dichtefunktion abweichen. Zudem lässt sich ein Q-Q-plot nutzen, um die empirischen Quantile der Residuen

mit jenen der Standardnormalverteilung zu vergleichen. Die Datenpunkte sollten hier auf einer Geraden liegen (Abb. 6.6, vgl. Abschn. 14.6.5). Für einen inferenzstatistischen Test mit der H_0 , dass Normalverteilung vorliegt, bietet sich jener nach Shapiro-Wilk an.

```
> hist(Estud, main="Histogramm stud. Residuen", freq=FALSE) # Histogramm

# Dichtefunktion der Standardnormalverteilung hinzufügen
> curve(dnorm(x, mean=0, sd=1), col="red", lwd=2, add=TRUE)
> qqnorm(Estud, main="Q-Q-Plot stud. Residuen") # Q-Q-plot
> qqline(Estud, col="red", lwd=2) # Referenzgerade für NV

# Shapiro-Wilk-Test auf Normalverteilung der studentisierten Residuen
> shapiro.test(Estud)
Shapiro-Wilk normality test
data: Estud
W = 0.9879, p-value = 0.4978
```

Soll eingeschätzt werden, ob die Annahme von Homoskedastizität plausibel ist, kann die bevorzugte Residuen-Variante auf der Ordinate gegen die Vorhersage auf der Abszisse abgetragen werden (*spread-level-plot*, Abb. 6.6).¹⁸ Die Datenpunkte sollten überall gleichmäßig um die 0-Linie streuen. Anhand desselben Diagramms kann auch die Unabhängigkeit der Messfehler heuristisch geprüft werden: Die Residuen sollten eine Verteilung aufweisen, die nicht systematisch mit der Vorhersage zusammenhängt.¹⁹

```
# studentisierte Residuen gegen Vorhersage darstellen
> plot(fitted(fitHAS), Estud, pch=20, xlab="Vorhersage",
+       ylab="studentisierte Residuen", main="Spread-Level-Plot")

> abline(h=0, col="red", lwd=2) # Referenz für Modellgültigkeit
```

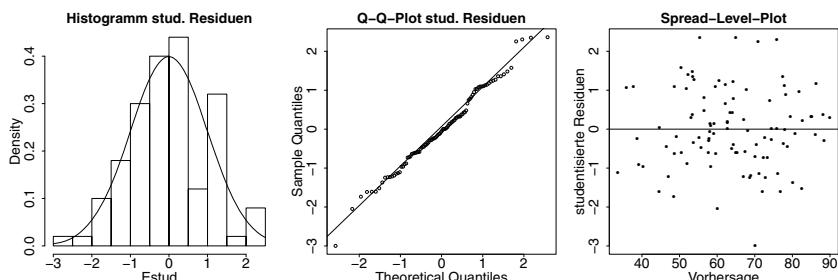


Abbildung 6.6: Grafische Prüfung der Verteilungsvoraussetzungen für eine Regressionsanalyse

Für manche Fälle, in denen die Daten darauf hindeuten, dass die Verteilung einer Variable Y nicht den Voraussetzungen genügt, können strenge monotone Transformationen die Verteilung

¹⁸Mitunter werden hierfür auch die Beträge der Residuen bzw. deren Wurzel gewählt (*scale-location plot*). Vergleiche weiterhin `residualPlots()` aus dem Paket `car`. Der Breusch-Pagan-Test auf Heteroskedastizität kann mit `bptest()` aus dem Paket `lmtest` (Zeileis & Hothorn, 2002) durchgeführt werden.

¹⁹Für den Durbin-Watson-Test auf Autokorrelation der Messfehler vgl. `durbinWatsonTest()` aus dem Paket `car`.

günstig beeinflussen, bei echt positiven Daten etwa Potenzfunktionen der Form Y^λ : Dazu zählen der Kehrwert Y^{-1} , der Logarithmus $\ln Y$ (per definitionem für $\lambda = 0$), die Quadratwurzel $Y^{\frac{1}{2}}$ (z. B. bei absoluten Häufigkeiten), oder der Arkussinus der Quadratwurzel $\arcsin Y^{\frac{1}{2}}$ (z. B. bei Anteilen). In der Regression finden häufig Box-Cox-Transformationen $\frac{Y^\lambda - 1}{\lambda}$ für $\lambda \neq 0$ bzw. $\ln Y$ für $\lambda = 0$ Verwendung, die ebenfalls echt positive Daten voraussetzen. Für sie stellt das Paket `car` die in Kombination miteinander zu verwendenden Funktionen `powerTransform()` und `bcPower()` bereit.

```
> powerTransform(⟨lm-Modell⟩, family="bcPower")
> bcPower(⟨Vektor⟩, ⟨lambda⟩)
```

Als erstes Argument von `powerTransform()` kann ein mit `lm()` erstelltes Modell angegeben werden. Für die Maximum-Likelihood-Schätzung des Parameters λ ist das Argument `family` auf "bcPower" zu setzen. λ erhält man aus dem zurückgegebenen Objekt durch `coef()`. Die Box-Cox-Transformation selbst führt `bcPower()` durch und benötigt dafür als Argument zum einen den Vektor der zu transformierenden Werte, zum anderen den Parameter λ der Transformation.

```
> library(car)                                # für boxCox(), powerTransform()
> lamObj <- powerTransform(fitHAS, family="bcPower")
> (lambda <- coef(lamObj))                   # max log-likelihood lambda
Y1
1.057033

# transformiertes Kriterium und manuelle Kontrolle
> wTrans <- bcPower(weight, coef(lambda))    # Transformation
> all.equal(wTrans, ((weight^lambda) - 1) / lambda)  # manuell
[1] TRUE
```

6.5.3 Multikollinearität

Multikollinearität liegt in einer multiplen Regression dann vor, wenn sich die Werte eines Prädiktors gut aus einer Linearkombination der übrigen Prädiktoren vorhersagen lassen. Dies ist insbesondere dann der Fall, wenn Prädiktoren paarweise miteinander korrelieren. Für die multiple Regression hat dies als unerwünschte Konsequenz einerseits weniger stabile Schätzungen der Koeffizienten zur Folge, die mit hohen Schätzfehlern versehen sind. Ebenso kann sich die Parameterschätzung bzgl. desselben Prädiktors stark in Abhängigkeit davon ändern, welche anderen Prädiktoren noch berücksichtigt werden. Andererseits ergeben sich Schwierigkeiten bei der Interpretation der b_j - bzw. der standardisierten b_j^z -Gewichte: Verglichen mit der Korrelation der zugehörigen Variable mit dem Kriterium können letztere unerwartet große oder kleine Werte annehmen und auch im Vorzeichen von der Korrelation abweichen.²⁰

Ob paarweise lineare Abhängigkeiten vorliegen, lässt sich anhand der Korrelationsmatrix R_x der Prädiktoren prüfen.

²⁰Auf numerischer Seite bringt starke Multikollinearität das Problem mit sich, dass die interne Berechnung der Parameterschätzungen anfälliger für Fehler werden kann, die aus der notwendigen Ungenauigkeit der Repräsentation von Gleitkommazahlen in Computern herrühren (vgl. Abschn. 1.3.6).

```
> (Rx <- cor(cbind(height, age, sport)))      # Korrelationsmatrix
      height        age        sport
height  1.00000000  0.06782798 -0.20937559
age    0.06782798  1.00000000  0.09496699
sport -0.20937559  0.09496699  1.00000000
```

Die Diagonalelemente der Inversen R_x^{-1} liefern den *Varianzinflationsfaktor* VIF_j jedes Prädiktors j als weitere Möglichkeit zur Kollinearitätsdiagnostik: $\sqrt{VIF_j}$ ist der Faktor, um den das Konfidenzintervall für das wahre β_j -Gewicht breiter als im analogen Fall linear unabhängiger Prädiktoren ist. VIF_j berechnet sich alternativ als $\frac{1}{1-R_j^2}$, also als Kehrwert der Toleranz $1 - R_j^2$, wobei R_j^2 der Determinationskoeffizient bei der Regression des Prädiktors j auf alle übrigen Prädiktoren ist. Aus dem Paket `car` stammt die Funktion `vif(lm-Modell)`, die als Argument ein durch `lm()` erstelltes lineares Modell erwartet.

```
> library(car)                                # für vif()
> vif(fitHAS)
      height        age        sport
1.054409  1.017361  1.059110
```

In der Ausgabe findet sich unter dem Namen jedes Prädiktors der zugehörige VIF_j -Wert. Da geringe Werte für die Toleranz auf lineare Abhängigkeit zwischen den Prädiktoren hindeuten, gilt dasselbe für große VIF-Werte. Konventionell werden VIF-Werte von bis zu ca. 4 als unkritisch, jene über 10 als starke Indikatoren für Multikollinearität gewertet. Es folgt die manuelle Kontrolle anhand der Regressionen jeweils eines Prädiktors auf alle übrigen.

```
# Regression jeweils eines Prädiktors auf alle übrigen
> fitHeight <- lm(height ~ age + sport)
> fitAge    <- lm(age    ~ height + sport)
> fitSport   <- lm(sport   ~ height + age)

# VIF_j aus zugehörigem Determinationskoeffizienten R^2
> 1 / (1 - summary(fitHeight)$r.squared)      # VIF height
[1] 1.054409

> 1 / (1 - summary(fitAge)$r.squared)          # VIF age
[1] 1.017361

> 1 / (1 - summary(fitSport)$r.squared)         # VIF sport
[1] 1.05911

# alternativ: Diagonalelemente der Inversen der Korrelationsmatrix
> diag(solve(Rx))
      height        age        sport
1.054409  1.017361  1.059110
```

Ein weiterer Kennwert zur Beurteilung von Multikollinearität ist die Kondition κ der Designmatrix X des meist mit standardisierten Variablen gebildeten linearen Modells (vgl. Abschn. 12.1.5, 12.9.1). Werte von $\kappa > 20$ sprechen einer Faustregel folgend für Multikollinearität. Zur

Berechnung dient `kappa(lm-Modell)`. Neben κ eignen sich zur differenzierteren Diagnose auch die Eigenwerte von $\mathbf{X}^t \mathbf{X}$ selbst sowie ihr jeweiliger Konditionsindex.²¹

```
# Regressionsmodell mit standardisierten Prädiktoren
> fitScl <- lm(scale(weight) ~ scale(height) + scale(age) + scale(sport))
> kappa(fitScl, exact=TRUE)
[1] 1.279833

> X      <- model.matrix(fitScl)           # Designmatrix
> (eigVals <- eigen(t(X) %*% X)$values)    # Eigenwerte von X^t * X
[1] 119.93081 103.85018 100.00000 73.21902

# Konditionsindizes: jeweils Wurzel aus Eigenwert / (Minimum != 0)
> sqrt(eigVals / min(eigVals[eigVals >= .Machine$double.eps]))
[1] 1.279833 1.190945 1.168660 1.000000
```

Wenn von den ursprünglichen Variablen zu zentrierten oder standardisierten Variablen übergegangen wird, ändern sich die VIF_j Werte nur dann, wenn multiplikative Terme, also Interaktionen in der Regression einbezogen sind (vgl. Abschn. 6.3.4). Dagegen ändert sich κ bei solchen Variablentransformationen praktisch immer.²²

```
# VIF: Regression mit standardisierten Variablen -> kein Unterschied
> vif(fitScl)
scale(height)  scale(age)  scale(sport)
1.054409     1.017361     1.059110

# kappa: Regression mit ursprünglichen Variablen -> Unterschied
> kappa(lm(weight ~ height + age + sport), exact=TRUE)
[1] 4804.947
```

6.6 Erweiterungen der linearen Regression

6.6.1 Robuste Regression

Die Parameterschätzung *robuster* Regressionsverfahren soll weniger sensibel auf die Verletzung von Voraussetzungen und die Anwesenheit von Ausreißern reagieren (Maronna, Martin & Yohai, 2006). Das Paket `robustbase` stellt für zwei Varianten der robusten Regression `lmrob()` und `ltsReg()` bereit. Eine Alternative liefert `r1m()` aus dem Paket MASS (Venables & Ripley, 2002). Einen Überblick über weitere Quellen gibt der Abschnitt *Robust Statistical Methods* der CRAN Task Views (Maechler, 2014).

²¹ Fortgeschrittene Methoden zur Diagnostik von Multikollinearität enthält das Paket `perturb` (Hendrickx, 2012).

²² Ursache dafür ist die Änderung der Eigenwerte bei Datentransformationen: Ist \mathbf{X} die Designmatrix des ursprünglichen Modells und \mathbf{X}' die Designmatrix des Modells der transformierten Daten, so gehen die Eigenwerte von $\mathbf{X}'^t \mathbf{X}'$ nicht auf einfache Weise aus denen von $\mathbf{X}^t \mathbf{X}$ hervor. Insbesondere verändern sich der größte und kleinste Eigenwert jeweils unterschiedlich, so dass deren Quotient nicht konstant ist.

```
> library(MASS) # für rlm()
> fitRLM <- rlm(weight ~ height + age + sport)
> summary(fitRLM)
Call: rlm(formula = weight ~ height + age + sport)
Residuals:
    Min      1Q  Median      3Q     Max 
-8.5957 -1.7949 -0.1092  2.1210  8.6398 

Coefficients:
            Value Std. Error t value
(Intercept) 14.1231    7.1223   1.9829
height       0.4895    0.0403  12.1514
age        -0.3436    0.0337 -10.1834
sport       -0.4181    0.0100 -41.9709

Residual standard error: 2.901 on 96 degrees of freedom
```

Allgemein können Bootstrap-Verfahren geeignet sein, trotz verletzter Modellvoraussetzungen angemessene Standardfehler der Parameterschätzungen zu erhalten (vgl. Abschn. 11.1.4).

Speziell für Schätzungen der Standardfehler der Regression unter Heteroskedastizität vgl. `sandwich(<lm-Modell>)` und `vcovHC(<lm-Modell>)` aus dem Paket `sandwich` (Zeileis, 2004). Diese Funktionen bestimmen die Kovarianzmatrix der Parameterschätzer eines mit `lm()` angepassten Modells neu. Wald-Tests der Parameter können dann mit `coeftest(<lm-Modell>, vcov=Schätzer)` aus dem Paket `lmtest` durchgeführt werden, wobei für das Argument `vcov` das Ergebnis von `sandwich()` oder `vcovHC()` anzugeben ist.

```
> library(sandwich) # für vcovHC()
> hcSE <- vcovHC(fitHAS, type="HC3")
> library(lmtest) # für coeftest()
> coeftest(fitHAS, vcov=hcSE)
t test of coefficients:

            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 8.101233  8.083794  1.0022  0.3188    
height       0.514893  0.046507 11.0714 < 2.2e-16 ***
age        -0.286646  0.042196 -6.7932 9.177e-10 ***
sport       -0.415952  0.011293 -36.8330 < 2.2e-16 ***
```

6.6.2 Penalisierte Regression

Bei hoher Multikollinearität und in Situationen mit sehr vielen Prädiktoren (gemessen an der Anzahl verfügbarer Beobachtungen), kommen *penalisierte* Regressionsverfahren in Betracht. Sie beschränken den geschätzten Parametervektor $\hat{\beta}$ ohne $\hat{\beta}_0$ in seiner Norm, reduzieren also den Betrag der Parameterschätzungen (*shrinkage*). Die resultierenden Parameterschätzungen sind nicht mehr unverzerrt, für den Preis eines geringen bias können sie jedoch eine deutlich geringere

Varianz der Schätzungen aufweisen als die übliche lineare Regression. Penalisierte Regressionen werden üblicherweise mit standardisierten Prädiktoren und Kriterium durchgeführt.

Die Ridge-Regression wird durch `lm.ridge()` aus dem MASS Paket bereitgestellt. Die Funktion akzeptiert eine Modellformel und für das Argument `lambda` den Wert des Tuning-Parameters λ . Dieser kontrolliert, wie stark das shrinkage ausfällt. Im ersten Schritt empfiehlt es sich, für `lambda` einen Vektor mit vielen Werten zu übergeben, etwa im Intervall [0.01, 10000]. `lm.ridge()` berechnet dann für jeden Wert von λ den Vorhersagefehler aus der verallgemeinerten Kreuzvalidierung (GCV, vgl. Abschn. 13.1.2 sowie Abb. 6.7).

```
> library(MASS) # für lm.ridge(), select()
> lambdas <- 10^(seq(-2, 4, length=100)) # Tuning-Parameter
> ridgeGCV <- lm.ridge(scale(weight) ~ scale(height) + scale(age) +
+ scale(sport), lambda=lambdas)
```

Im Anschluss erfährt man mit `select(<lm.ridge-Objekt>)` für welchen Wert von λ der Kreuzvalidierungsfehler sein Minimum erreicht.

```
> select(ridgeGCV)
modified HKB estimator is 0.06879525
modified L-W estimator is 0.06014114
smallest value of GCV at 0.2205131
```

In einem erneuten Aufruf von `lm.ridge()` kann schließlich `lambda` auf den vorher identifizierten Wert mit dem geringsten Kreuzvalidierungsfehler gesetzt werden. Aus dem erzeugten Objekt extrahiert `coef()` dann die Parameterschätzungen.

```
> ridgeSel <- lm.ridge(scale(weight) ~ scale(height) + scale(age) +
+ scale(sport), lambda=0.22)

> coef(ridgeSel) # Parameterschätzungen
scale(height) scale(age) scale(sport)
-3.766000e-16 2.744047e-01 -1.707210e-01 -8.478151e-01
```

Die Methoden LASSO und elastic net wählen anders als die Ridge-Regression gleichzeitig auch eine Teilmenge von Prädiktoren aus, indem sie abhängig von λ einzelne Parameterschätzungen auf 0 reduzieren. Eine Umsetzung von ridge, LASSO und elastic net liefern `cv.glmnet()` sowie `glmnet()` aus dem Paket `glmnet` (Friedman, Hastie & Tibshirani, 2010).

Anders als die bisher vorgestellten Funktionen akzeptieren `cv.glmnet()` und `glmnet()` keine Modellformel. Stattdessen muss für `x` die Matrix der Prädiktoren übergeben werden und für `y` der Vektor der modellierten Zielvariable. Dabei berechnet die Funktion `cv.glmnet()` für von ihr selbst gewählte Werte von λ den Vorhersagefehler aus der verallgemeinerten Kreuzvalidierung für `nFolds` viele Partitionen (vgl. Abschn. 13.1.2). Die zurückgegebene Liste speichert in der Komponente `lambda.min` den Wert für λ , der den Kreuzvalidierungsfehler minimiert. Für die Ridge-Regression ist das Argument `alpha=0` zu setzen. Der Regularisierungspfad – hier der Verlauf der Parameterschätzungen $\hat{\beta}_j$ in Abhängigkeit von $\ln \lambda$ – ist in Abb. 6.7 dargestellt.

```
# Matrix der standardisierten Prädiktoren und des Kriteriums
> matScl <- scale(cbind(weight, height, age, sport))
```

```
> library(glmnet) # für cv.glmnet(), glmnet()

# verallgemeinerte Kreuzvalidierung für lambda - Ridge-Regression
> ridgeCV <- cv.glmnet(matScl[ , c("height", "age", "sport")],
+ matScl[ , "weight"], nfolds=10, alpha=0)
```

Anschließend passt `glmnet()` das Modell für einen konkreten Wert von λ an und gibt ein Objekt zurück, aus dem sich mit `coef()` die Parameterschätzungen extrahieren lassen.

```
> ridge <- glmnet(matScl[ , c("height", "age", "sport")],
+ matScl[ , "weight"], lambda=ridgeCV$lambda.min, alpha=0)

> coef(ridge) # Parameterschätzungen
(Intercept) -3.837235e-16
height       2.640929e-01
age          -1.618367e-01
sport         -7.806060e-01
```

Für das LASSO-Verfahren ist in `cv.glmnet()` und `glmnet()` das Argument `alpha=1` zu setzen. Der Regularisierungspfad – hier der Verlauf der Parameterschätzungen in Abhängigkeit von der L_1 -Norm $\sum_j |\hat{\beta}_j|$ des Vektors $\hat{\beta}$ – ist in Abb. 6.7 dargestellt.

```
# verallgemeinerte Kreuzvalidierung für lambda - LASSO
> lassoCV <- cv.glmnet(matScl[ , c("height", "age", "sport")],
+ matScl[ , "weight"], nfolds=10, alpha=1)

> lasso <- glmnet(matScl[ , c("height", "age", "sport")],
+ matScl[ , "weight"], lambda=lassoCV$lambda.min, alpha=1)

> coef(lasso) # Parameterschätzungen
(Intercept) -3.886238e-16
height       2.708041e-01
age          -1.667786e-01
sport         -8.466213e-01
```

Für das elastic-net-Verfahren ist in `cv.glmnet()` und `glmnet()` das Argument `alpha=0.5` zu setzen. Der Regularisierungspfad – hier der Verlauf der Parameterschätzungen $\hat{\beta}_j$ in Abhängigkeit vom Anteil der aufgeklärten Devianz – ist in Abb. 6.7 dargestellt.

```
# verallgemeinerte Kreuzvalidierung für lambda - elastic net
> elNetCV <- cv.glmnet(matScl[ , c("height", "age", "sport")],
+ matScl[ , "weight"], nfolds=10, alpha=0.5)

> elNet <- glmnet(matScl[ , c("height", "age", "sport")],
+ matScl[ , "weight"], lambda=lassoCV$lambda.min, alpha=0.5)

> coef(elNet) # Parameterschätzungen
(Intercept) -3.918004e-16
height       2.724987e-01
```

```
age      -1.686541e-01
sport    -8.464099e-01
```

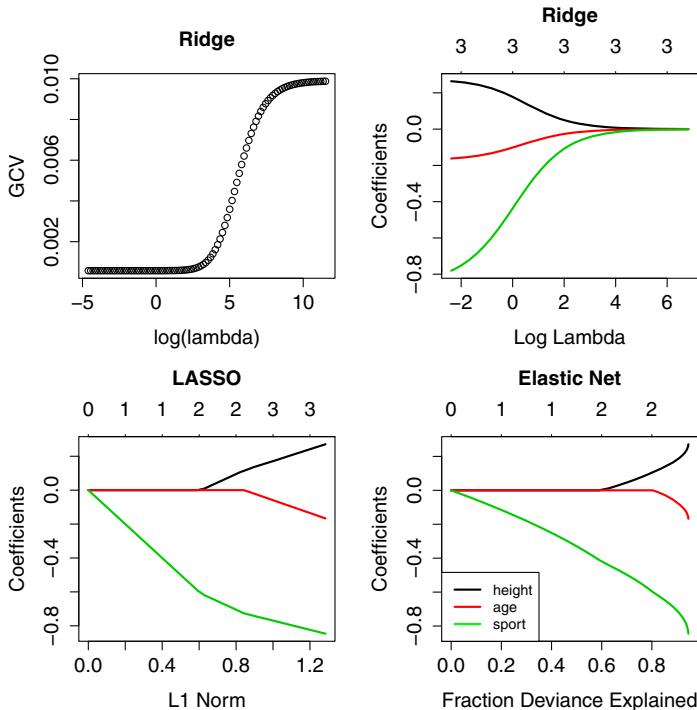


Abbildung 6.7: Penalisierte Regression: Kreuzvalidierungsfehler in Abhängigkeit von $\ln \lambda$ in der Ridge-Regression mit `lm.ridge()`. Regularisierungspfade für Ridge-Regression, LASSO und elastic net mit `glmnet()`. Die obere x-Achse zeigt die Anzahl ausgewählter Parameter $\hat{\beta}_j \neq 0$.

6.6.3 Nichtlineare Zusammenhänge

Ein Regressionsmodell, das linear in den Parametern ist, kann dennoch das Kriterium als nichtlineare Funktion einzelner Prädiktoren beschreiben. So erleichtert es die Funktion `poly(<Prädiktor>, degree=<Grad>)`, orthogonale Polynome für eine polynomiale Regression mit quadratischen Termen oder Polynomen höheren Grades zu erstellen. Analog stellt das im Basisumfang von R enthaltene Paket `splines` mit `ns(<Prädiktor>, df=<Grad>)` natürliche splines (außerhalb des beobachteten Wertebereichs linear) und mit `bs(<Prädiktor>, df=<Grad>)` B-splines (mit Bernstein-Polynomen als Basis-Funktionen) zur Verfügung. Beide Funktionen können innerhalb der Modellformel von `lm()` verwendet werden und besitzen das Argument `df`, um ihre Variabilität zu kontrollieren.

Verallgemeinerte additive Modelle werden durch die Funktion `gam()` aus dem Paket `mgcv` (Wood, 2006) unterstützt. Sie verwendet die verallgemeinerte Kreuzvalidierung, um automatisch die Variabilitätsparameter der eingesetzten splines auszuwählen.

Für die Bestimmung von Parametern in nichtlinearen Vorhersagemodellen anhand der Methode der kleinsten quadrierten Abweichungen vgl. `nls()` sowie Ritz und Streibig (2009).

6.6.4 Abhängige Fehler bei Messwiederholung oder Clusterung

Liefert eine Person mehrere Beobachtungen zu unterschiedlichen Zeitpunkten, sind die einzelnen Messwerte nicht mehr unabhängig voneinander. Dies ist auch der Fall, wenn verschiedene Personen aufgrund eines gemeinsamen Merkmals cluster bilden, z. B. durch eine familiäre Verwandtschaft. Eine wesentliche Voraussetzung der linearen Regression ist dann verletzt. Sowohl gemischte Modelle (*linear mixed effects models*) als auch verallgemeinerte Schätzgleichungen (*generalized estimating equations*, GEE) kommen für abhängige Daten in Betracht. Beide Herangehensweisen sind auch für kategoriale Zielvariablen geeignet und verallgemeinern so etwa die logistische Regression oder die Poisson-Regression (vgl. Kap. 8).

Gemischte Modelle (Long, 2012; Pinheiro & Bates, 2000; West, Welch & Gałecki, 2014) können Abhängigkeitsstrukturen von Daten flexibel berücksichtigen. Sie werden von den Paketen `lme4` (Bates, Maechler, Bolker & Walker, 2014) sowie `nlme` (Pinheiro, Bates, DebRoy, Sarkar & R Core Team, 2014) unterstützt und eignen sich auch für Situationen, in denen nicht alle Personen dieselbe Anzahl von Messwerten zu denselben Zeitpunkten liefern. Dabei ist es möglich, kategoriale und kontinuierliche Prädiktoren einzubeziehen, die auch zeitabhängig, d. h. innerhalb einer Person veränderlich sein dürfen. In varianzanalytischen Designs mit Messwiederholung (vgl. Abschn. 7.4, 7.6, 7.7) bieten gemischte Modelle häufig eine alternative Möglichkeit zur Auswertung, die die Abhängigkeitsstruktur der Messungen besser als die klassische Varianzanalyse berücksichtigt.

Verallgemeinerte Schätzgleichungen (GEE) modellieren die Abhängigkeit des bedingten Erwartungswerts der Zielvariable von den Prädiktoren getrennt von der Abhängigkeitsstruktur der Daten derselben Person oder desselben clusters. GEE sind im Paket `geepack` (Højsgaard, Halekoh & Yan, 2006; Yan & Fine, 2004) implementiert.

6.7 Partialkorrelation und Semipartialkorrelation

Ein von `lm()` erzeugtes Objekt kann auch zur Ermittlung der Partialkorrelation $r_{(XY),Z}$ zweier Variablen X und Y ohne eine dritte Variable Z verwendet werden (vgl. Abschn. 2.7.8): $r_{(XY),Z}$ ist gleich der Korrelation der Residuen der Regressionen jeweils von X und Y auf Z .

```
# Partialkorrelation weight mit height, beide ohne sport
> weight.S <- residuals(lm(weight ~ sport))    # Residuen: weight ohne sport
> height.S <- residuals(lm(height ~ sport))    # Residuen: height ohne Sport
> cor(weight.S, height.S)
[1] 0.6595016
```

Zur Kontrolle lässt sich die konventionelle Formel umsetzen oder der Umstand nutzen, dass Partialkorrelationen aus der invertierten Korrelationsmatrix \mathbf{R}^{-1} der drei beteiligten Variablen berechnet werden können: Ist \mathbf{D} die Diagonalmatrix aus den Diagonalelementen von \mathbf{R}^{-1} , stehen die Partialkorrelationen außerhalb der Diagonale der Matrix $-(\mathbf{D}^{-\frac{1}{2}} \mathbf{R}^{-1} \mathbf{D}^{-\frac{1}{2}})$ (vgl. Abschn. 12.1 für Multiplikation und Invertierung von Matrizen).

```
# Kontrolle über herkömmliche Formel
> (cor(weight, height) - (cor(weight, sport) * cor(height, sport))) /
+   sqrt((1-cor(weight, sport)^2) * (1-cor(height, sport)^2))
[1] 0.6595016
```

```
# Kontrolle über Inverse der Korrelationsmatrix
> R      <- cor(cbind(weight, height, sport))      # Korrelationsmatrix
> Rinv   <- solve(R)                                # ihre Inverse R^(-1)
> DsqrtInv <- diag(1/sqrt(diag(Rinv)))           # D^(-1/2)
> Rpart   <- -(DsqrtInv %*% Rinv %*% DsqrtInv)
> Rpart[upper.tri(Rpart)]                          # alle Partialkorrelationen
[1] 0.6595016 -0.9468826 0.5738578
```

Gleichzeitig beinhaltet $-(\mathbf{D}^{-1} \mathbf{R}^{-1})$ außerhalb der Diagonale die standardisierten b_j^z -Gewichte der Regressionen mit jeweils dem durch die Zeile definierten Kriterium und den durch die Spalten definierten Prädiktoren.

```
> Dinv <- diag(1/diag(Rinv))                      # D^(-1)
> bz   <- -(Dinv %*% Rinv)                        # -(D^(-1) * R^(-1))
> bz[upper.tri(bz)]                               # alle bz-Gewichte
[1] 0.2589668 -0.8691255 1.3414126
```

```
# standardisierte Regression weight auf height und sport
> fit <- lm(scale(weight) ~ scale(height) + scale(sport))
> zapsmall(coef(fit))                            # bz-Gewichte
(Intercept) scale(height) scale(sport)
0.0000000 0.2589668 -0.8691255
```

Die Semipartialkorrelation $r_{(X,Z)Y} = \frac{r_{XY} - (r_{XZ} \cdot r_{YZ})}{\sqrt{1 - r_{XZ}^2}}$ einer Variable Y mit einer Variable X ohne Z lässt sich analog zur Partialkorrelation als Korrelation von Y mit den Residuen der Regression von X auf Z berechnen.

```
# Semipartialkorrelation weight mit height ohne sport
> cor(weight, height.S)
[1] 0.2532269
```

```
# Kontrolle über herkömmliche Formel
> (cor(weight, height) - (cor(weight, sport) * cor(height, sport))) /
+   sqrt(1-cor(height, sport)^2)
[1] 0.2532269
```

Partialkorrelation und Semipartialkorrelation lassen sich auf die Situation verallgemeinern, dass nicht nur eine Variable Z auspartialisiert wird, sondern mehrere Variablen Z_j . Die Partialkorre-

lation $r_{(XY).Z_j}$ ist dann die Korrelation der Residuen der multiplen Regression von X auf alle Z_j mit den Residuen der multiplen Regression von Y auf alle Z_j . Ebenso ist die Semipartialkorrelation $r_{(X.Z_j)Y}$ die Korrelation von Y mit den Residuen der multiplen Regression von X auf alle Z_j .

```
# Residuen Regression von weight bzw. height auf verbleibende Variablen
> weight.AS <- residuals(lm(weight ~ age + sport))
> height.AS <- residuals(lm(height ~ age + sport))

# Partialkorrelation (weight mit height) ohne verbleibende Variablen
> (pcorWH.AS <- cor(weight.AS, height.AS))
[1] 0.7531376

# Semipartialkorrelation weight mit (height ohne verbleibende Variablen)
> (spcorWH.AS <- cor(weight, height.AS))
[1] 0.2674680
```

Die Semipartialkorrelation $r_{(X.Z_j)Y}$ weist folgenden Zusammenhang zur multiplen Regression des Kriteriums Y auf die Prädiktoren X und Z_j auf: $r_{(X.Z_j)Y}^2$ ist gleich der Erhöhung des Determinationskoeffizienten R^2 beim Übergang von der Regression von Y auf Z_j hin zur Regression von Y auf die Prädiktoren X und Z_j . In diesem Sinne ist die quadrierte Semipartialkorrelation des Kriteriums mit einem Prädiktor ohne alle übrigen gleich dem Varianzanteil, den der Prädiktor zusätzlich zu allen übrigen Prädiktoren aufklärt.

```
> spcorWH.AS^2                      # quadrierte Semipartialkorrelation
[1] 0.07153911

# R^2 Differenz: Regressionen mit bzw. ohne height inkl. übriger Prädiktoren
> fitAS  <- lm(weight ~ age + sport)           # ohne height
> fitHAS <- lm(weight ~ height + age + sport)   # mit height
> summary(fitHAS)$r.squared - summary(fitAS)$r.squared
[1] 0.07153911
```

Ist b_x das Regressionsgewicht des Prädiktors X in der Regression des Kriteriums Y auf die Prädiktoren X und Z_j , und ist weiterhin $X.Z_j$ die Variable der Residuen der Regression von X auf alle Z_j , so gilt analog zur einfachen linearen Regression $b_x = \frac{K_{(X.Z_j)Y}}{s_{X.Z_j}^2}$, wenn K für die Kovarianz und s^2 für die Varianz von Variablen steht.

```
> fitHAS                      # multiple Regression
Call:
lm(formula = weight ~ height + age + sport)

Coefficients:
(Intercept)    height      age      sport
     8.1012    0.5149   -0.2866   -0.4160

# b-Gewicht von height
```

Kapitel 6 Lineare Regression

```
> cov(weight, height.AS) / var(height.AS)
[1] 0.5148935
```

Kapitel 7

***t*-Tests und Varianzanalysen**

Häufig bestehen in empirischen Untersuchungen Hypothesen über Erwartungswerte von Variablen. Viele der für solche Hypothesen geeigneten Tests gehen davon aus, dass bestimmte Annahmen über die Verteilungen der Variablen erfüllt sind, dass etwa in allen Bedingungen Normalverteilungen mit derselben Varianz vorliegen. Bevor auf Tests zum Vergleich von Erwartungswerten eingegangen wird, sollen deshalb zunächst jene Verfahren vorgestellt werden, die sich mit der Prüfung statistischer Voraussetzungen befassen (vgl. auch Abschn. 10.1). Für die statistischen Grundlagen dieser Themen vgl. Eid et al. (2013); Maxwell und Delaney (2004).

7.1 Tests auf Varianzhomogenität

Die ab Abschn. 7.2.2 und 7.3 vorgestellten Tests auf Erwartungswertunterschiede setzen voraus, dass die Variable in allen Bedingungen normalverteilt mit derselben Varianz ist. Um zu prüfen, ob die erhobenen Werte mit der Annahme von Varianzhomogenität konsistent sind, existieren verschiedene Testverfahren, bei deren Anwendung der Umstand zu berücksichtigen ist, dass i. d. R. die H_0 den gewünschten Zustand darstellt. Um die power eines Tests zu vergrößern, wird mitunter ein höher als übliches α -Niveau in der Größenordnung von 0.2 gewählt.

7.1.1 *F*-Test auf Varianzhomogenität für zwei Stichproben

Um festzustellen, ob die empirischen Varianzen einer Variable in zwei unabhängigen Stichproben mit der H_0 verträglich sind, dass die Variable in beiden Bedingungen dieselbe theoretische Varianz besitzt, kann die Funktion `var.test()` benutzt werden. Diese vergleicht beide empirischen Varianzen mittels eines *F*-Tests, der aber sensibel auf Verletzungen der Voraussetzung reagiert, dass die Variable in den Bedingungen normalverteilt ist.¹

```
> var.test(x=(Vektor), y=(Vektor), conf.level=0.95,  
+           alternative=c("two.sided", "less", "greater"))
```

Unter `x` und `y` sind die Daten aus beiden Stichproben einzutragen. Alternativ zu `x` und `y` kann auch eine Modellformel $\langle AV \rangle \sim \langle UV \rangle$ angegeben werden. Dabei ist $\langle UV \rangle$ ein Faktor mit zwei Ausprägungen und derselben Länge wie $\langle AV \rangle$, der für jede Beobachtung die Gruppenzugehörigkeit codiert. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Die Argumente `alternative` und `conf.level` beziehen sich auf

¹ Als Alternative für den Fall nicht normalverteilter Variablen existieren die nonparametrischen Tests nach Mood bzw. Ansari-Bradley (vgl. Abschn. 10.4).

die Größe des *F*-Bruchs unter H_1 im Vergleich zu seiner Größe unter H_0 (1) bzw. auf die Breite seines Vertrauensintervalls.

```
> n1 <- 110                                # Gruppengröße 1
> n2 <- 90                                  # Gruppengröße 2
> DV1 <- rnorm(n1, mean=100, sd=15)          # Daten Gruppe 1
> DV2 <- rnorm(n2, mean=100, sd=13)          # Daten Gruppe 2
> DV <- c(DV1, DV2)                          # Gesamt-Daten
> IV <- factor(rep(c("A", "B"), c(n1, n2))) # Gruppierungsfaktor
> var.test(DV ~ IV)

F test to compare two variances
data: DV by IV
F = 1.6821, num df = 109, denom df = 89, p-value = 0.01157
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
1.124894 2.495117
sample estimates:
ratio of variances
1.682140
```

Die Ausgabe des Tests umfasst den empirischen *F*-Wert (*F*), der gleich dem Quotienten der zu testenden Varianzen ist (**ratio of variances**). Zusammen mit den zugehörigen Freiheitsgraden (UV-Effekt: `num df`, Fehler: `denom df`) wird der *p*-Wert (`p-value`) ausgegeben sowie das Konfidenzintervall für das Verhältnis der Varianzen in der gewünschten Breite. Das Ergebnis lässt sich manuell bestätigen:

```
> var1 <- var(DV1)                           # korrig. Varianz Gruppe 1
> var2 <- var(DV2)                           # korrig. Varianz Gruppe 2
> Fval <- var1 / var2                       # Teststatistik
> (pVal <- 2*pf(Fval, n1-1, n2-1, lower.tail=FALSE)) # p-Wert
[1] 0.0115746

# zweiseitiges 95%-Vertrauensintervall für den Quotienten der Varianzen
> Fcrit1 <- qf(0.05/2, n1-1, n2-1, lower.tail=FALSE) # krit. Wert rechts
> Fcrit2 <- qf(0.05/2, n1-1, n2-1)                  # kritischer Wert links
> (ciLo <- var1 / (Fcrit1*var2))                   # untere Grenze
[1] 1.124894

> (ciUp <- var1 / (Fcrit2*var2))                   # obere Grenze
[1] 2.495117
```

7.1.2 Levene-Test für mehr als zwei Stichproben

Ein Test auf Varianzhomogenität für mehr als zwei Gruppen ist jener nach Levene, für den das Paket `car` benötigt wird. Der Levene-Test reagiert robust auf Verletzungen der Voraussetzung von Normalverteiltheit.

```
> leveneTest(y=<Daten>, group=<Faktor>, data=<Datensatz>)
```

Die Daten *y* können in Form eines Vektors zusammen mit einer zugehörigen Gruppierungsvariable *group* als Objekt der Klasse **factor** derselben Länge wie *y* angegeben werden. Alternativ ist dies als Modellformel $\langle\text{AV}\rangle \sim \langle\text{UV}\rangle$ oder als lineares Modell möglich, wie es **lm()** als Objekt zurückgibt. Unter *data* ist ein Datensatz anzugeben, wenn die verwendeten Variablen aus einem Datensatz stammen. In einer Modellformel verwendete Variablen müssen immer aus einem Datensatz kommen.

```
> Nj <- c(22, 18, 20)                                # Gruppengrößen
> N  <- sum(Nj)                                     # Gesamt-N
> P  <- length(Nj)                                   # Anzahl Gruppen
> DV <- sample(0:100, N, replace=TRUE)                # AV Daten

# Personen zufällig auf Gruppen aufteilen
> IV    <- factor(sample(rep(1:P, Nj), N, replace=FALSE))
> levDf <- data.frame(IV, DV)                         # Datensatz

> library(car)                                       # für leveneTest()
> leveneTest(DV ~ IV, data=levDf)
Levene's Test for Homogeneity of Variance
  Df   F value Pr(>F)
IV   2    2.113  0.1302
57
```

Da der Levene-Test letztlich eine Varianzanalyse ist, beinhaltet die Ausgabe einen empirischen *F*-Wert (*F value*) mit den Freiheitsgraden von Effekt- und Residual-Quadratsumme (*Df*) sowie den zugehörigen *p*-Wert (*Pr(>F)*). In der beim Levene-Test durchgeführten Varianzanalyse gehen statt der ursprünglichen Werte der AV die jeweiligen Beträge ihrer Differenz zum zugehörigen Gruppenmedian ein.² Der Test lässt sich so auch manuell durchführen (vgl. Abschn. 7.3).

```
# Berechnung mit den absoluten Abweichungen zum Median jeder Gruppe
> absDiff <- abs(DV - ave(DV, IV, FUN=median))      # Betrag Abweichungen
> anova(lm(absDiff ~ IV))                            # Varianzanalyse ...

# alternativ: absolute Abweichungen zum Gruppenmittelwert
> absErr <- abs(DV - ave(DV, IV, FUN=mean))        # Betrag Residuen
> anova(lm(absErr ~ IV))                            # Varianzanalyse ...
```

7.1.3 Fligner-Killeen-Test für mehr als zwei Stichproben

Der Fligner-Killeen-Test auf Varianzhomogenität für mehr als zwei Gruppen basiert auf den Rängen der absoluten Abweichungen der Daten zu ihrem Gruppenmedian. Er verwendet eine asymptotisch χ^2 -verteilte Teststatistik und gilt als robust gegenüber Verletzungen der Voraussetzung von Normalverteiltheit.

²Diese Variante wird auch als Brown-Forsythe-Test bezeichnet. Mit **leveneTest(..., center=mean)** können alternativ die Differenzen zum jeweiligen Gruppenmittelwert gewählt werden.

```
> fligner.test(x=<Vektor>, g=<Faktor>, data=<Datensatz>)
```

Unter **x** wird der Datenvektor eingetragen und unter **g** die zugehörige Gruppierungsvariable. Sie ist ein Objekt der Klasse **factor** derselben Länge wie **x** und gibt für jeden Wert in **x** an, aus welcher Bedingung er stammt. Alternativ zu **x** und **g** kann eine Modellformel $\langle AV \rangle \sim \langle UV \rangle$ verwendet und ggf. unter **data** ein Datensatz angegeben werden, aus dem die in der Modellformel verwendeten Variablen stammen.

```
> fligner.test(DV ~ IV, data=levDf)
Fligner-Killeen test of homogeneity of variances
data: DV by IV
Fligner-Killeen:med chi-squared = 4.5021, df = 2, p-value = 0.1053
```

```
# manuell: Standard-NV-Quantile der transformierten Ränge in der
# Gesamtstichprobe der absoluten Abweichungen zum Gruppenmedian
> absDiff <- abs(DV - ave(DV, IV, FUN=median))      # Betrag Abweichungen
> quants  <- qnorm((0.5 + rank(absDiff) / (2*(N+1))), 0, 1)
> MQj     <- tapply(quants, IV, mean)      # mittlere Quantile pro Gruppe

# asymptotisch chi^2 verteilte Teststatistik
> (FK <- sum(Nj * ((MQj - mean(quants))^2)) / var(quants))
[1] 4.502134

> (pVal <- pchisq(FK, P-1, lower.tail=FALSE))      # p-Wert
[1] 0.1052868
```

7.2 *t*-Tests

Hypothesen über den Erwartungswert einer Variable in einer oder zwei Bedingungen lassen sich mit *t*-Tests prüfen. Für die analogen multivariaten T^2 -Tests mit mehreren AVn vgl. Abschn. 12.6.

7.2.1 *t*-Test für eine Stichprobe

Der einfache *t*-Test prüft, ob die in einer Stichprobe ermittelten Werte einer normalverteilten Variable mit der H_0 verträglich sind, dass diese Variable einen bestimmten Erwartungswert μ_0 besitzt.

```
> t.test(x=<Vektor>, alternative=c("two.sided", "less", "greater"),
+         mu=0, conf.level=0.95)
```

Unter **x** ist der Datenvektor einzutragen. Mit **alternative** wird festgelegt, ob die H_1 gerichtet oder ungerichtet ist. "less" und "greater" beziehen sich dabei auf die Reihenfolge Erwartungswert unter H_1 "less" bzw. "greater" μ_0 . Das Argument **mu** bestimmt μ_0 . **conf.level** legt die Breite des je nach H_1 ein- oder zweiseitigen Konfidenzintervalls für den Erwartungswert μ fest.

```
> N      <- 100                      # Stichprobengröße
> DV    <- rnorm(N, 5, 20)          # AV
> muH0 <- 0                         # Erwartungswert unter H0
> t.test(DV, alternative="two.sided", mu=muH0)
One Sample t-test
data: DV
t = 2.4125, df = 99, p-value = 0.01769
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
0.8989476 9.2292614
sample estimates:
mean of x
5.064104
```

Die Ausgabe umfasst den empirischen *t*-Wert (*t*) mit den Freiheitsgraden (*df*) und dem zugehörigen *p*-Wert (*p-value*). Weiterhin wird das Konfidenzintervall für μ in der gewünschten Breite sowie der Mittelwert genannt. Das Ergebnis lässt sich manuell verifizieren:

```
> M      <- mean(DV)                # Mittelwert
> s      <- sd(DV)                 # korrig. Streuung
> (tVal <- (M-muH0) / (s/sqrt(N))) # Teststatistik t
[1] 2.412462

> (pVal <- 2*pt(tVal, N-1, lower.tail=FALSE)) # p-Wert
[1] 0.01768618

# zweiseitiges 95%-Vertrauensintervall für mu
> tCrit <- qt(0.05/2, N-1, lower.tail=FALSE) # kritischer t-Wert
> (ciLo <- M - tCrit * s/sqrt(N))           # untere Grenze
[1] 0.8989476

> (ciUp <- M + tCrit * s/sqrt(N))           # obere Grenze
[1] 9.229261
```

Als Effektstärkemaß kann Cohens δ herangezogen werden, dessen Schätzung *d* auf der Differenz des Mittelwerts zum Erwartungswert unter H_0 beruht, die an der korrigierten Streuung relativiert wird.³

```
> (d <- (mean(DV) - muH0) / sd(DV))        # Schätzung Cohens d
[1] 0.2412462
```

³Das Paket MBESS (Kelley & Lai, 2012) enthält eigene Funktionen, um viele der hier manuell geschätzten Effektstärken inkl. eines Vertrauensintervalls zu berechnen. Für die hier verwendeten Formeln vgl. Eid et al. (2013).

7.2.2 *t*-Test für zwei unabhängige Stichproben

Im *t*-Test für unabhängige Stichproben werden die in zwei unabhängigen Stichproben ermittelten Werte einer normalverteilten Variable daraufhin miteinander verglichen, ob sie mit der H_0 verträglich sind, dass die Variable in den zugehörigen Bedingungen denselben Erwartungswert besitzt.

```
> t.test(x=<Vektor>, y=<Vektor>, paired=FALSE, conf.level=0.95,
+         alternative=c("two.sided", "less", "greater"), var.equal=FALSE)
```

Unter **x** sind die Daten der ersten Stichprobe einzutragen, unter **y** entsprechend die der zweiten. Alternativ zu **x** und **y** kann auch eine Modellformel $\langle AV \rangle \sim \langle UV \rangle$ angegeben werden. Dabei ist $\langle UV \rangle$ ein Faktor mit zwei Ausprägungen und derselben Länge wie $\langle AV \rangle$, der für jede Beobachtung die Gruppenzugehörigkeit codiert. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter **data** zu nennen. Mit **alternative** wird festgelegt, ob die H_1 gerichtet oder ungerichtet ist. "less" und "greater" beziehen sich dabei auf die Reihenfolge **x** "less" bzw. "greater" **y**. Mit dem Argument **paired** wird bestimmt, ob es sich um unabhängige (FALSE) oder abhängige (TRUE) Stichproben handelt. **var.equal** gibt an, ob von Varianzhomogenität in den beiden Bedingungen ausgegangen werden soll (Voreinstellung ist FALSE). Das von **conf.level** in seiner Breite festgelegte Vertrauensintervall bezieht sich auf die Differenz der Erwartungswerte und ist je nach H_1 ein- oder zweiseitig.

Test mit Annahme von Varianzhomogenität und Schätzung der Effektstärke

Als Beispiel soll die Körpergröße von Männern und Frauen betrachtet werden. Die Fragestellung ist gerichtet – getestet werden soll, ob der Erwartungswert bei den Männern größer als jener bei den Frauen ist. Zunächst sei Varianzhomogenität vorausgesetzt.

```
> n1  <- 18                                # Stichprobenumfang 1
> n2  <- 21                                # Stichprobenumfang 2
> DVm <- rnorm(n1, 180, 10)                 # Daten Männer
> DVf <- rnorm(n2, 175, 6)                  # Daten Frauen
> DV  <- c(DVm, DVf)                      # Gesamt-Daten
> IV  <- factor(rep(0:1, c(n1, n2)), labels=c("m", "f")) # Grupp.Faktor
> t.test(DV ~ IV, alternative="greater", var.equal=TRUE)
Two Sample t-test
data: DV ~ IV
t = 1.6346, df = 37, p-value = 0.05531
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
-0.1201456      Inf
sample estimates:
mean in group m mean in group f
180.1867        176.4451
```

Beim Verwendung einer Modellformel $\langle AV \rangle \sim \langle UV \rangle$ ist für gerichtete Tests darauf zu achten, dass sich die Reihenfolge der Gruppen über die Reihenfolge der Faktorstufen in $\langle UV \rangle$ bestimmt.

```
# manuelle Kontrolle: gepoolte Streuung
> sdPool <- sqrt(((n1-1)*var(DVm) + (n2-1)*var(DVf)) / (n1+n2-2))

# Schätzung der Streuung der Mittelwertsdifferenz
> estSigDiff <- sqrt((n1+n2) / (n1*n2)) * sdPool
> (tVal      <- (mean(DVm)-mean(DVf)) / estSigDiff)           # t-Wert
[1] 1.634606

> (pVal <- pt(tVal, n1+n2-2, lower.tail=FALSE))    # einseitiger p-Wert
[1] 0.05530639

# untere Grenze des einseitigen 95%-Vertrauensintervalls für mu
> tCrit <- qt(1-0.05, n1+n2-2)          # zweiseitig: qt(1-(0.05/2), n1+n2-2)
> (ciLo <- mean(DVm)-mean(DVf) - tCrit*estSigDiff)
[1] -0.1201456
```

Als Effektstärkemaß kann Cohens δ herangezogen werden. Seine Schätzung d beruht auf der Mittelwertsdifferenz, die an der gepoolten Streuung relativiert wird, wie sie auch in der *t*-Statistik Verwendung findet.

```
> (d <- (mean(DVm) - mean(DVf)) / sdPool)           # Schätzung Cohens d
[1] 0.5250485
```

Test ohne Annahme von Varianzhomogenität

Ohne Voraussetzung von Varianzhomogenität verwendet R die als Welch-Test bezeichnete Variante des *t*-Tests, deren andere Berechnung der Teststatistik sowie der Freiheitsgrade zu einem i. a. etwas konservativeren Test führt.

```
> t.test(DV ~ IV, alternative="greater", var.equal=FALSE)
Welch Two Sample t-test
data: DV by IV
t = 1.5681, df = 25.727, p-value = 0.06454
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
-0.3298099      Inf
sample estimates:
mean in group m mean in group f
180.1867        176.4451

# manuelle Kontrolle
> varM      <- var(DVm)                      # korrigierte Varianz Männer
> varF      <- var(DVf)                      # korrigierte Varianz Frauen
> num       <- (varM/n1 + varF/n2)^2         # Zähler
> denom     <- varM^2/((n1-1)*n1^2) + varF^2/((n2-1)*n2^2)   # Nenner
> (dfWelch <- num/denom)                      # Freiheitsgrade im Welch-Test
[1] 25.72683
```

```
> (tValW <- (mean(DVm)-mean(DVf)) / sqrt(varM/n1 + varF/n2)) # t-Wert
[1] 1.568068

> (pValW <- pt(tValW, dfWelch, lower.tail=FALSE)) # einseitiger p-Wert
[1] 0.06454212
```

7.2.3 *t*-Test für zwei abhängige Stichproben

Der *t*-Test für abhängige Stichproben prüft, ob die Erwartungswerte einer in zwei Bedingungen paarweise erhobenen Variable identisch sind. Er wird wie jener für unabhängige Stichproben durchgeführt, jedoch ist hier das Argument `paired=TRUE` für `t.test()` zu verwenden. Der Test setzt voraus, dass sich die in `x` und `y` angegebenen Daten einander paarweise zuordnen lassen, weshalb `x` und `y` dieselbe Länge besitzen müssen. Der Test prüft die H_0 , dass die sich aus paarweiser Subtraktion ergebende Differenzvariable von `x` und `y` den Erwartungswert 0 besitzt. Entsprechend bezieht sich auch das Konfidenzintervall auf den Erwartungswert dieser Differenzvariable.

```
> N      <- 20                      # Stichprobengröße
> DVpre <- rnorm(N, mean=90,  sd=15)    # Daten prä
> DVpost <- rnorm(N, mean=100, sd=15)    # Daten post
> DV     <- c(DVpre, DVpost)          # Gesamt-Daten

# Faktor, der jedem Wert von DV Messzeitpunkt zuordnet
# dabei Kontrolle der Reihenfolge der Faktorstufen: pre vor post
> IV <- factor(rep(0:1, each=N), labels=c("pre", "post"))
> t.test(DV ~ IV, alternative="less", paired=TRUE)

Paired t-test
data: DV by IV
t = -2.0818, df = 19, p-value = 0.02556
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
-Inf -1.790952
sample estimates:
mean of the differences
-10.57236

# äquivalent: 1-Stichproben t-Test der Differenzvariable
> DVdiff <- DVpre-DVpost            # Differenzvariable
> t.test(DVdiff, alternative="less")  # ...
```

Als Effektstärkemaß kann Cohens δ herangezogen werden, dessen Schätzung d mit der Differenzvariable wie im Fall für eine Stichprobe vollzogen wird, wobei $\mu_0 = 0$ gilt.

```
> (d <- mean(DVdiff) / sd(DVdiff))           # Schätzung Cohens d
[1] -0.4655015
```

7.3 Einfaktorielle Varianzanalyse (CR-*p*)

Bestehen Hypothesen über die Erwartungswerte einer Variable in mehr als zwei Bedingungen, wird häufig eine Varianzanalyse (ANOVA, *analysis of variance*) zur Prüfung herangezogen. Die einfaktorielle Varianzanalyse ohne Messwiederholung mit *p* Gruppen (CR-*p*, *completely randomized design*⁴) verallgemeinert dabei die Fragestellung eines *t*-Tests für unabhängige Stichproben auf Situationen, in denen Werte einer normalverteilten Variable in mehr als zwei Gruppen ermittelt werden. Für die analoge multivariate Varianzanalyse mit mehreren AVn vgl. Abschn. 12.7.1.

Durch die enge Verwandtschaft von linearer Regression und Varianzanalyse (vgl. Abschn. 12.9 für eine formalere Darstellung) ähneln sich die Befehle für beide Analysen in R häufig stark. Zur Unterscheidung ist die Art der Variablen auf der rechten Seite der Modellformel bedeutsam (vgl. Abschn. 5.2): Im Fall der Regression sind dies quantitative Prädiktoren (numerische Vektoren), im Fall der Varianzanalyse dagegen kategoriale Gruppierungsvariablen, also Objekte der Klasse **factor**. Damit R auch bei Gruppierungsvariablen mit numerisch codierten Stufen die richtige Interpretation als Faktor und nicht als quantitativer Prädiktor vornehmen kann, ist darauf zu achten, dass die UVn tatsächlich die Klasse **factor** besitzen.

7.3.1 Auswertung mit `oneway.test()`

In der einfaktoriellen Varianzanalyse wird geprüft, ob die in verschiedenen Bedingungen erhöhen Werte einer Variable mit der H_0 verträglich sind, dass diese Variable in allen Gruppen denselben Erwartungswert besitzt. Die H_1 ist unspezifisch und lautet, dass sich mindestens zwei Erwartungswerte unterscheiden.

```
> oneway.test(formula=<Modellformel>, data=<Datensatz>, subset=<Indexvektor>,
+               var.equal=FALSE)
```

Unter **formula** sind Daten und Gruppierungsvariable als Modellformel $\langle\text{AV}\rangle \sim \langle\text{UV}\rangle$ einzugeben, wobei $\langle\text{UV}\rangle$ ein Faktor derselben Länge wie $\langle\text{AV}\rangle$ ist und für jede Beobachtung in $\langle\text{AV}\rangle$ die zugehörige UV-Stufe angibt. Geschieht dies mit Variablen aus einem Datensatz, muss dieser unter **data** eingetragen werden. Das Argument **subset** erlaubt es, nur eine Teilmenge der Fälle einfließen zu lassen, es erwartet einen entsprechenden Indexvektor, der sich auf die Zeilen des Datensatzes bezieht. Mit **var.equal** wird vorgegeben, ob von Varianzhomogenität ausgegangen werden kann (Voreinstellung ist FALSE).

```
> P      <- 4                                # Anzahl Gruppen
> Nj    <- c(41, 37, 42, 40)                 # Gruppengrößen
> IVeff <- c(0, 0.3, 0.6, 1)                # Gruppen-Effekte
> IV    <- factor(rep(LETTERS[1:P], Nj))    # Gruppierungsfaktor
> DV    <- IVeff[unclass(IV)] + rnorm(sum(Nj), 0, 1)  # Gesamt-AV
> dfCRp <- data.frame(IV, DV)                # Datensatz
> oneway.test(DV ~ IV, data=dfCRp, var.equal=TRUE)
One-way analysis of means
```

⁴Hier und im folgenden wird für varianzanalytische Versuchspläne die von Kirk (2013) eingeführte Notation übernommen.

```
data: DV and IV
F = 4.8619, num df = 3, denom df = 156, p-value = 0.002922
```

Die Ausgabe von `oneway.test()` beschränkt sich auf die wesentlichen Ergebnisse des Hypothesentests: Dies sind der empirische *F*-Wert (*F*) mit den Freiheitsgraden der Effekt-Quadratsumme (`num df`) im Zähler (numerator) des *F*-Bruchs und jener der Quadratsumme der Residuen (`denom df`) im Nenner (denominator) des *F*-Bruchs sowie der zugehörige *p*-Wert (`p-value`).

Für ausführlichere Informationen zum Modell, etwa zu den Quadratsummen von Effekt und Residuen, sollte auf `aov()` oder `anova()` zurückgegriffen werden (vgl. Abschn. 7.3.2 bzw. 7.3.3). Die Ergebnisse können auch manuell überprüft werden.

```
> N <- sum(Nj)                                     # Gesamt-N
> Vj <- tapply(DV, IV, var)                      # korrig. Gruppenvarianzen
> Mj <- tapply(DV, IV, mean)                      # Gruppenmittel
> M <- sum((Nj/N) * Mj)                           # gewichtetes Gesamtmittel

# Quadratsumme Residuen, alternativ: sum((DV - ave(DV, IV, FUN=mean))^2)
> SSw   <- sum((Nj-1) * Vj)                      # Quadratsumme within
> SSb   <- sum(Nj * (Mj-M)^2)                     # Quadratsumme between
> MSw   <- SSw / (N-P)                           # mittlere QS within
> MSb   <- SSb / (P-1)                            # mittlere QS between
> (Fval <- MSb / MSw)                            # Teststatistik F-Wert
[1] 4.861867

> (pVal <- pf(Fval, P-1, N-P, lower.tail=FALSE)) # p-Wert
[1] 0.002921932
```

Ist von Varianzhomogenität nicht auszugehen und deshalb das Argument `var.equal=FALSE` gesetzt, führt `oneway.test()` einen auf mehr als zwei Stichproben verallgemeinerten Welch-Test durch (vgl. Abschn. 7.2.2), der i. a. etwas konservativer ist – im Beispiel allerdings einen kleineren *p*-Wert liefert.

```
> oneway.test(DV ~ IV, data=dfCRp, var.equal=FALSE)
One-way analysis of means (not assuming equal variances)
data: DV and IV
F = 5.3235, num df = 3.000, denom df = 85.576, p-value = 0.002065
```

7.3.2 Auswertung mit `aov()`

Wenn Varianzhomogenität anzunehmen ist, kann eine Varianzanalyse auch mit `aov()` berechnet werden.

```
> aov(formula=<Modellformel>, data=<Datensatz>, subset=<Indexvektor>)
```

Unter `formula` werden Daten und Gruppierungsvariable als Modellformel $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$ einge tragen, wobei $\langle \text{UV} \rangle$ ein Faktor derselben Länge wie $\langle \text{AV} \rangle$ ist und für jede Beobachtung in $\langle \text{AV} \rangle$ die zugehörige UV-Stufe angibt. Unter `data` ist ggf. der Datensatz als Quelle der Variablen

anzugeben. Das Argument `subset` erlaubt es, nur eine Teilmenge der Fälle einfließen zu lassen, es erwartet einen entsprechenden Indexvektor, der sich auf die Zeilen des Datensatzes bezieht.

Das von `aov()` zurückgegebene Objekt ist wie das Ergebnis von `lm()` eine Liste, die u. a. die berechneten Quadratsummen und Freiheitsgrade enthält.⁵ Um von UV-Effekt und Residuen auch ihre inferenzstatistischen Größen zu erhalten, muss `summary()` auf das Ergebnis angewendet werden.

```
> aovCRp <- aov(DV ~ IV, data=dfCRp)
> summary(aovCRp)
    Df   Sum Sq  Mean Sq  F value    Pr(>F)
IV       3   13.782   4.5941   4.8619  0.002922 ***
Residuals 156 147.407   0.9449
```

Die Kennwerte von UV-Effekt (Zeile IV) und Residuen (Zeile `Residuals`) finden sich in den Spalten `Df` (Freiheitsgrade), `Sum Sq` (Quadratsumme), `Mean Sq` (mittlere Quadratsumme), `F value` (Teststatistik: *F*-Wert) und `Pr(>F)` (*p*-Wert).

Eine tabellarische Übersicht über den Gesamtmittelwert, die Mittelwerte in den einzelnen Bedingungen und deren Zellbesetzung erzeugt `model.tables()`.

```
> model.tables(<aov-Objekt>, type="Schätzer", se=FALSE)
```

Im ersten Argument muss das Ergebnis einer durch `aov()` vorgenommenen Modellanpassung stehen. Über das Argument `type` wird bestimmt, ob die Mittelwerte ("means") oder geschätzten Effektgrößen (Voreinstellung "effects") ausgegeben werden sollen. Letztere erhält man auch mit `coef(<aov-Objekt>)`, für ihre Bedeutung vgl. Abschn. 12.9.2. Sollen Standardfehler genannt werden, ist `se=TRUE` zu setzen.

```
> model.tables(aovCRp, type="means")
Tables of means
Grand mean
0.4362142
```

	A	B	C	D
IV	0.2080	0.07393	0.6395	0.7917
rep	41.0000	37.00000	42.0000	40.0000

Grafisch aufbereitet werden deskriptive Kennwerte der AV in den einzelnen Gruppen mit `plot.design()`. In der Voreinstellung sind dies die Mittelwerte, über das Argument `fun` lassen sich jedoch durch Übergabe einer geeigneten Funktion auch beliebige andere Kennwerte berechnen. Die Anwendung von `plot.design()` ist insbesondere sinnvoll, wenn mehr als ein Faktor als UV variiert wird (vgl. Abschn. 7.5.1, Abb. 7.3).

```
> plot.design(<Modellformel>, fun=mean, data=<Datensatz>)

> plot.design(DV ~ IV, fun=mean, data=dfCRp,
+             main="Mittelwerte getrennt nach Gruppen")
```

⁵Da `aov()` letztlich `lm()` aufruft, lassen sich auf diese Liste dieselben Funktionen zur Extraktion weiterer Informationen anwenden (vgl. Abschn. 6.2).

7.3.3 Auswertung mit anova()

Äquivalent zum Aufruf von `summary(aov-Modell)` ist die Verwendung der `anova()` Funktion, wenn sie auf ein mit `lm()` formuliertes lineares Modell angewendet wird. Das Ergebnis von `anova(lm-Modell)` unterscheidet sich auf der Konsole nur wenig von der Ausgabe von `summary(aov-Modell)`. `anova()` gibt jedoch einen Datensatz zurück, aus dem sich besonders einfache Freiheitsgrade, Quadratsummen und *p*-Werte für weitere Rechnungen extrahieren lassen. Dagegen ist die von `summary(aov-Modell)` zurückgegebene Liste deutlich komplexer aufgebaut.

```
> (anovaCRp <- anova(lm(DV ~ IV, data=dfCRp)))
Analysis of Variance Table
Response: DV
  Df  Sum Sq  Mean Sq  F value    Pr(>F)
IV      3   13.782   4.5941   4.8619  0.002922 **
Residuals 156  147.407   0.9449

> anovaCRp["Residuals", "Sum Sq"]          # Residual-Quadratsumme
[1] 147.4069
```

Analog zur Verwendung von `anova()` zum Vergleich zweier unterschiedlich umfassender Regressionsmodelle (vgl. Kap. 6.3.3) lässt sich auch die Varianzanalyse als Vergleich von *nested* Modellen durchführen: Das eingeschränkte Modell `(fitR)` berücksichtigt hier als Effektterm nur den für alle Beobachtungen konstanten Erwartungswert ($\langle AV \rangle \sim 1$), das umfassendere Modell `(fitU)` zusätzlich den Gruppierungsfaktor ($\langle AV \rangle \sim \langle UV \rangle$). Der Modellvergleich erfolgt dann mit `anova(fitR, fitU)` (vgl. Abschn. 12.9.6).

```
> fitR <- lm(DV ~ 1, data=dfCRp)           # eingeschränktes Modell
> fitU <- lm(DV ~ IV, data=dfCRp)         # umfassendes Modell
> anova(fitR, fitU)                      # Modellvergleich
Analysis of Variance Table
Model 1: DV ~ 1
Model 2: DV ~ IV
  Res.Df   RSS  Df  Sum of Sq     F    Pr(>F)
1     159 161.19
2     156 147.41  3     13.782  4.8619  0.002922 **
```

7.3.4 Effektstärke schätzen

Als Maß für die Effektstärke wird oft η^2 herangezogen, in dessen Schätzung $\hat{\eta}^2 = \frac{SS_b}{SS_b + SS_w}$ die Quadratsumme von Effekt (SS_b) und Residuen (SS_w) einfließen. Die Berechnung – inkl. des hier identischen partiellen $\hat{\eta}_p^2$ – übernimmt `EtaSq(aov-Objekt)` aus dem Paket `DescTools`.

```
> library(DescTools)                      # für EtaSq()
> EtaSq(aovCRp, type=1)
  eta.sq  eta.sq.part
IV 0.08550311  0.08550311
```

Ein ähnliches Maß ist ω^2 , dessen Schätzung $\hat{\omega}^2 = \frac{df_b \cdot (MS_b - MS_w)}{SS_b + SS_w + MS_w}$ auch die mittleren Quadratsummen von Effekt (MS_b) und Residuen (MS_w) sowie die Freiheitsgrade df_b von SS_b einbezieht. Alternativ eignet sich $\hat{f} = \sqrt{\frac{\hat{\eta}^2}{1-\hat{\eta}^2}}$.

```
> dfSSb <- anovaCRp["IV",           "Df"]          # df QS between
> SSb   <- anovaCRp["IV",           "Sum Sq"]      # Quadratsumme between
> MSb   <- anovaCRp["IV",           "Mean Sq"]     # mittlere QS between
> SSw   <- anovaCRp["Residuals", "Sum Sq"]      # Quadratsumme within
> MSw   <- anovaCRp["Residuals", "Mean Sq"]    # mittlere QS within

# Schätzungen verschiedener Effektstärken
> (etaSq <- SSb / (SSb + SSw))                # Schätzung von eta^2
[1] 0.08550311

> (omegaSq <- dfSSb * (MSb-MSw) / (SSb+SSw+MSw)) # Schätzung von omega^2
[1] 0.06752082

> (f <- sqrt(etaSq / (1-etaSq)))                 # Schätzung von f
[1] 0.3057735
```

7.3.5 Voraussetzungen grafisch prüfen

Eine Varianzanalyse setzt u. a. voraus, dass die Fehler unabhängige, normalverteilte Variablen mit Erwartungswert 0 und fester Streuung sind.⁶ Ob empirische Daten einer Stichprobe mit diesen Annahmen konsistent sind, kann heuristisch durch eine Reihe von Diagrammen abgeschätzt werden, wie sie auch in der Regressionsdiagnostik Verwendung finden (vgl. Abschn. 6.5). Dazu können die (ggf. studentisierten) Residuen gegen die Gruppenmittelwerte abgetragen werden – bei Modellgültigkeit sollten sie unabhängig vom Gruppenmittelwert zufällig um 0 streuen. Ebenso kann die empirische Verteilung der Residuen in jeder Gruppe mit Hilfe von boxplots veranschaulicht werden (vgl. Abschn. 14.6.3) – diese Verteilungen sollten einander ähnlich sein. Schließlich lassen sich die Residuen mittels eines Quantil-Quantil-Plots danach beurteilen, ob sie mit der Annahme von Normalverteiltheit der Fehler verträglich sind (Abb. 7.1, vgl. Abschn. 14.6.5).

```
# studentisierte Residuen gegen Vorhersage darstellen
> Estud <- rstudent(aovCRp)           # studentisierte Residuen
> plot(fitted(aovCRp), Estud, pch=20, ylab="stud. Residuen",
+       xlab="Vorhersage", main="stud. Residuen gegen Vorhersage")

> abline(h=0, col="gray60", lwd=2)      # Referenz für Modellgültigkeit

# Verteilung der studentisierten Residuen in den Gruppen
> plot(Estud ~ aovCRp$model$IV, main="Residuen vs. Stufen")
```

⁶Für alternative Verfahren, die robuster gegenüber der Verletzung bestimmter Voraussetzungen sind, oder weniger Voraussetzungen machen, vgl. Abschn. 6.6.1, 10.5.6, 11.1.5.

```
# Normalverteiltheit der Fehler prüfen
> qqnorm(Estud, pch=20) # Q-Q-Plot der Residuen
> qqline(Estud, col="gray60") # Referenzgerade
```

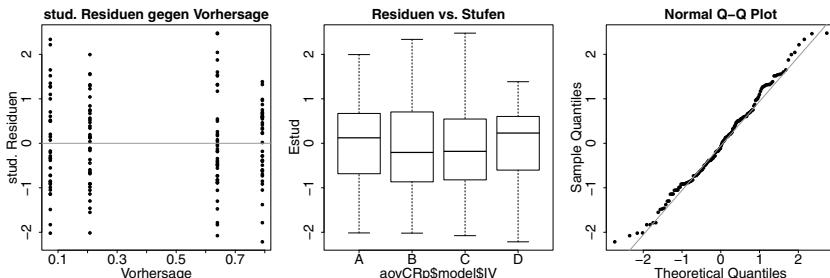


Abbildung 7.1: Grafische Prüfung der Voraussetzungen für die einfaktorielle Varianzanalyse

7.3.6 Einzelvergleiche (Kontraste)

Ein Kontrast $\psi = \sum_j c_j \mu_j$ meint im folgenden eine Linearkombination der p Gruppenerwartungswerte μ_j mit den Koeffizienten c_j , wobei $1 \leq j \leq p$ sowie $\sum_j c_j = 0$ gilt. Solche Kontraste dienen einem spezifischen Vergleich zwischen den p experimentellen Bedingungen. Über die Größe von ψ lassen sich Hypothesen aufstellen (unter H_0 gilt meist $\psi_0 = 0$), deren Test im folgenden beschrieben wird (für eine formalere Darstellung vgl. Abschn. 12.9.5).

Beliebige a-priori Kontraste

Zum Testen beliebiger Kontraste stellt der Basisumfang von R keine spezialisierte Funktion bereit. Der Test lässt sich jedoch mit `glht()` (*general linear hypothesis test*) des Pakets `multcomp` (Hothorn, Bretz & Westfall, 2008) durchführen. Hierfür ist zunächst der Kontrastvektor c aus den Koeffizienten c_j der Linearkombination zu bilden.

```
> glht(<aov-Modell>, linfct=mcp(<UV>=<Kontrastkoeffizienten>),
+       alternative=c("two.sided", "less", "greater"))
```

Als erstes Argument ist ein mit `aov()` erstelltes Modell zu übergeben, dessen Gruppen einem spezifischen Vergleich unterzogen werden sollen. Dieser Vergleich kann mit dem `linfct` Argument definiert werden, wozu die `mcp()` Funktion dient. Diese erwartet ihrerseits eine Zuweisung der Form `<UV>=<Koeffizienten>` als Argument. Dabei ist `<UV>` der Name der UV aus dem mit `aov()` erstellten Modell (ohne Anführungszeichen) und `<Koeffizienten>` eine zeilenweise aus (ggf. benannten) Kontrastvektoren zusammengestellte Matrix. Dabei muss jeder Koeffizient c_j angegeben werden, also auch solche, die 0 sind und somit nicht in die Linearkombination eingehen. Mit `alternative` wird festgelegt, ob die H_1 gerichtet oder ungerichtet ("`two.sided`") ist. "`less`" und "`greater`" beziehen sich dabei auf die Reihenfolge $\psi < \psi_0$ bzw. $\psi > \psi_0$, wobei die Voreinstellung $\psi_0 = 0$ ist.

`summary(glht-Modell, test=adjusted("alpha-Adjustierung"))` testet das Ergebnis auf Signifikanz. Das Argument `test` erlaubt für den Test mehrerer Kontraste gleichzeitig, eine Methode zur α -Adjustierung auszuwählen (vgl. `?summary.glht`). Soll dies unterbleiben, ist `test=adjusted("none")` zu setzen. Durch `confint(glht-Modell)` erhält man die Vertrauensintervalle der definierten ψ , die sich über `plot(glht-Modell)` grafisch darstellen lassen. Beim Aufstellen von Kontrasten ist zu beachten, dass die Reihenfolge der Gruppen durch die Reihenfolge der Faktorstufen bestimmt wird (vgl. Abschn. 2.6.4).

Im Beispiel soll zunächst nur der Vergleich des Mittels der ersten beiden gegen das Mittel der verbleibenden Gruppen getestet werden. Dabei sei $\psi_0 = 0$ und unter $H_1 \psi < 0$.

```
# Matrix der Kontrastkoeffizienten - hier nur eine Zeile
> cntrMat <- rbind("(A+B)-(C+D)"=c(1/2, 1/2, -1/2, -1/2))
> library(multcomp)                                     # für glht()
> summary(glht(aovCRp, linfct=mcp(IV=cntrMat), alternative="less"))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = DV ~ IV, data = dfCRp)
Linear Hypotheses:
Estimate Std. Error t value Pr(<t)
(A+B)-(C+D) >= 0 -0.5746    0.1539 -3.735 0.000132 ***

```

Die Ausgabe führt den Namen des getesteten Kontrastes aus der Matrix der Kontrastkoeffizienten auf, gefolgt von der Schätzung $\hat{\psi}$, für die die Erwartungswerte in der Linearkombination durch die Gruppenmittelwerte ersetzt werden ($\hat{\psi} = \sum_j c_j M_j$). Es folgt der Standardfehler dieser Schätzung (Std. Error), der Wert der *t*-Teststatistik (t value) und der zugehörige *p*-Wert (Pr(<t)).

Die Differenz $\hat{\psi} - \psi_0$ bildet den Zähler der Teststatistik *t*. Im Nenner wird die quadrierte Länge $\|\mathbf{c}\|^2$ des Kontrastvektors benötigt, für dessen Berechnung eine Gewichtung mit den Zellbesetzungen vorzunehmen ist. Das Produkt $\|\mathbf{c}\|^2 \cdot MS_w$ der quadrierten Länge mit der mittleren Quadratsumme der Residuen aus der zugehörigen Varianzanalyse bildet den quadrierten Nenner der Teststatistik und stellt die Schätzung der Varianz des Kontrasts dar. Die Teststatistik ist im Fall von a-priori Kontrasten unter H_0 zentral *t*-verteilt mit den Freiheitsgraden der Quadratsumme der Residuen in der zugehörigen Varianzanalyse. Das α -Niveau betrage 0.05.

```
> P      <- nlevels(dfCRp$IV)                      # Anzahl der Gruppen
> Mj     <- tapply(dfCRp$DV, dfCRp$IV, mean)       # Gruppenmittelwerte
> Nj     <- table(dfCRp$IV)                         # Gruppengrößen
> dfSSw  <- sum(Nj) - P                            # df von SS within
> SSw    <- sum((DV - ave(DV, IV, FUN=mean))^2)   # SS within
> MSw    <- SSw / dfSSw                            # MS within
> (psiHat <- sum(cntrMat[1, ] * Mj))              # Kontrastschätzung
[1] -0.5746347

> lenSq  <- sum(cntrMat[1, ]^2 / Nj)                # quadrierte Länge
> (tStat <- psiHat / sqrt(lenSq*MSw))               # Teststatistik t
[1] -3.734508
```

```
> (tCrit <- qt(0.05, dfSSw, lower.tail=FALSE))      # krit. t-Wert eins.
[1] 1.65468

> (pVal <- pt(abs(tStat), dfSSw, lower.tail=FALSE)) # p-Wert einseitig
[1] 0.0001316091

# obere Grenze des einseitigen 95%-Vertrauensintervalls für psi
> (ciUp <- psiHat + tCrit*sqrt(lenSq*MSw))
[1] -0.3200264
```

Sollen mehrere Kontraste gleichzeitig getestet werden, verfügt die an das Argument `linfct` übergebene Funktion `mcp()` zum einen über Voreinstellungen für verschiedene Spezialfälle: Mit `mcp(⟨UV⟩="Dunnett")` erhält man etwa alle $p - 1$ paarweisen Vergleiche, in denen ein Gruppenerwartungswert gegen jenen der Referenzstufe (die erste Faktorstufe von `⟨UV⟩`, vgl. Abschn. 2.6.5) getestet wird. Analog erzeugt `mcp(⟨UV⟩="Tukey")` alle Tests der paarweisen Gruppenvergleiche (s. u.). Zum anderen sind allgemeine Kontrast-Tests durch Zusammenstellung der zugehörigen Kontrastvektoren als (ggf. benannte) Zeilen einer Matrix möglich. Hier werden drei Kontraste ohne Adjustierung des α -Niveaus gerichtet getestet.

```
> cntrMat <- rbind("A-D"           =c( 1,   0,   0, -1),
+                     "1/3*(A+B+C)-D"=c(1/3, 1/3, 1/3, -1),
+                     "B-C"           =c( 0,   1,  -1,  0))

> glhtRes <- glht(aovCRp, linfct=mcp(IV=cntrMat), alternative="less")
> plot(glhtRes)                 # Diagramm der Konfidenzintervalle
> (sumRes <- summary(glhtRes, test=adjusted("none")))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = DV ~ IV, data = dfCRp)
Linear Hypotheses:
Estimate Std. Error t value Pr(<t)
A-D >= 0          -0.5837    0.2160  -2.702 0.00383 **
1/3*(A+B+C)-D >= 0 -0.4845    0.1775  -2.729 0.00354 **
B-C >= 0          -0.5656    0.2192  -2.581 0.00539 **

# manuelle Berechnung
> psiHats <- cntrMat  %*% Mj          # Kontrastschätzungen
> lenSqs  <- cntrMat^2 %*% (1/Nj)       # quadrierte Längen
> tStats  <- psiHats / sqrt(lenSqs*MSw) # Teststatistiken
> pVals   <- pt(abs(tStats), dfSSw, lower.tail=FALSE) # p-Werte einseitig
> data.frame(psiHats, tStats, pVals)
  psiHats     tStats      pVals
A-D      -0.5836704 -2.701783 0.003830077
1/3*(A+B+C)-D -0.4845486 -2.729212 0.003539043
B-C      -0.5655991 -2.580620 0.005391977
```

Beliebige post-hoc Kontraste nach Scheffé

Beliebige Kontraste können auch im Anschluss an eine signifikante Varianzanalyse getestet werden. Die Varianzanalyse prüft implizit simultan alle möglichen Kontraste – spezifische Hypothesen liegen also bei ihrer Anwendung nicht vor. Aus diesem Grund muss im Anschluss bei Einzeltests eine geeignete α -Adjustierung vorgenommen werden, hier vorgestellt nach der Methode von Scheffé.⁷ Sie ist in `ScheffeTest()` aus dem Paket `DescTools` implementiert.

```
> ScheffeTest(<aov-Modell>, which="UV", contrasts=<Kontrastmatrix>,
+               conf.level=0.95)
```

Als erstes Argument ist die mit `aov()` angepasste ANOVA zu übergeben. `which` erwartet den Namen der UV, auf den sich die Kontraste beziehen, was hier im einfaktoriellen Fall optional ist. Für `contrasts` ist die Kontrastmatrix anzugeben, wobei die Koeffizienten eines Kontrasts in einer Spalte stehen. Das Argument `conf.level` legt die Breite des Konfidenzintervalls für ψ fest. Hier sollen wieder die drei Kontraste des letzten Abschnitts getestet werden.

```
> library(DescTools)                                     # für ScheffeTest()
# transponiere cntrMat mit t(), da Koeffizienten in einer Zeile stehen
> ScheffeTest(aovCRp, which="IV", contrasts=t(cntrMat))
Posthoc multiple comparisons of means : Scheffe Test
95% family-wise confidence level
Fit: aov(formula = DV ~ IV, data = dfCRp)
$IV
      diff     lwr.ci     upr.ci    pval
A-D   -0.5836704 -1.194230  0.02688944 0.0671 .
A,B,C-D -0.4845486 -0.986326  0.01722875 0.0629 .
B-C    -0.5655991 -1.185034  0.05383580 0.0880 .
```

In der Ausgabe stehen die Ergebnisse für jeweils einen Kontrast in einer Zeile. Dies sind die Kontrastschätzung $\hat{\psi}$ in der Spalte `diff`, das Konfidenzintervall für ψ in den Spalten `lwr.ci` und `upr.ci` und schließlich den *p*-Wert in der Spalte `pval`.

Bei der manuellen Kontrolle gilt zunächst alles bereits für a-priori Kontraste Ausgeführt. Lediglich die Wahl des kritischen Wertes weicht ab und ergibt sich zur α -Adjustierung aus einer *F*-Verteilung. Dieser kritische Wert ist mit der quadrierten a-priori *t*-Teststatistik zu vergleichen – die etwa in der von `summary(glht(...))` zurückgegebenen Liste in der Komponente `test$tstat` steht.

```
> dfSSB <- P-1                                         # df von SS between
> (Fstat <- sumRes$test$tstat^2)                         # quadr. t-Teststatistik
          A-D 1/3*(A+B+C)-D           B-C
          7.299631       7.448600      6.659600

# kritischer F-Wert für quadrierte t-Teststatistik
> (Fcrit <- dfSSB*qf(0.05, df1=dfSSB, df2=dfSSW, lower.tail=FALSE))
[1] 7.987706
```

⁷Für weitere vgl. `PostHocTest()` aus dem Paket `DescTools`.

```
# p-Wert einseitig
> (pVal <- pf(Fstat/dfSSb, dfSSb, dfSSw), lower.tail=FALSE)
      A-D 1/3*(A+B+C)-D          B-C
  0.06706707    0.06294284    0.08801514
```

Die Wahl des kritischen Wertes erfolgte hier so, dass alle möglichen Kontraste zugelassen sind. Sollen sich die Kontraste dagegen nur auf einen Teil der Erwartungswerte beziehen (und damit aus einem Unterraum des $(p - 1)$ -dimensionalen Kontrastraumes stammen), kann der kritische Wert entsprechend anders gewählt werden. In diesem Fall erfolgt eine gleichzeitige α -Adjustierung nur für Kontraste aus dem gewählten Unterraum, woraus ein etwas geringerer kritischer Wert resultiert. Ist q mit $q < p$ die Anzahl der relevanten Gruppen, wäre der kritische F -Wert $(q-1) * qf(1-0.05, q-1, dfSSw)$.

Paarvergleiche mit *t*-Tests und α -Adjustierung

Taucht die spezifischere Frage auf, welche paarweisen Gruppenunterschiede vorliegen, besteht eine andere Herangehensweise in der Anwendung von $\binom{p}{2} = \frac{p(p-1)}{2}$ *t*-Tests, um die Unterschiede jeweils zweier Gruppen auf Signifikanz zu prüfen. Das α -Niveau ist zu adjustieren, da sonst mehrfach die Möglichkeit bestünde, denselben Fehler erster Art zu begehen und das tatsächliche α -Niveau über dem nominellen läge. Anstatt mehrere solcher *t*-Tests manuell durchzuführen, lässt sich die `pairwise.t.test()` Funktion nutzen, die alle möglichen Paarvergleiche testet und verschiedene Verfahren zur α -Adjustierung anbietet.

```
> pairwise.t.test(x=<Vektor>, g=<Faktor>, p.adjust.method="holm",
+                   paired=FALSE, pool.sd=!paired, alternative="two.sided")
```

Der Vektor `x` muss die Daten der AV enthalten, `g` ist der zugehörige Faktor derselben Länge wie `x`, der für jedes Element von `x` codiert, in welcher Bedingung der Wert erhoben wurde. Über das Argument `p.adjust.method` wird die α -Adjustierung gesteuert, als Methoden stehen u. a. jene nach Holm (Voreinstellung) und Bonferroni zur Auswahl, für weitere vgl. `?p.adjust`. Ist von Varianzhomogenität auszugehen und daher `pool.sd=TRUE` gesetzt, wird die Fehlerstreuung auf Basis aller, also nicht nur anhand der beiden beim jeweiligen *t*-Test beteiligten Gruppen bestimmt. Dies ist jedoch nur möglich, wenn keine abhängigen Stichproben vorliegen, was mit dem Argument `paired` anzudeuten ist. Für gerichtete Tests ist das Argument `alternative` auf "less" oder "greater" zu setzen – Voreinstellung ist "two.sided" für ungerichtete Tests.

```
> DV <- dfCRp$DV                                # Daten
> IV <- dfCRp$IV                                # Gruppierungsfaktor
```

```
# paarweise t-Tests mit alpha-Adjustierung nach Bonferroni
> pairwise.t.test(DV, IV, p.adjust.method="bonferroni")
Pairwise comparisons using t tests with pooled SD
data: DV and IV
   A      B      C
B 1.0000 -     -
C 0.2694 0.0647 -
```

```
D 0.0460 0.0088 1.0000
P value adjustment method: bonferroni
```

Die in Form einer Matrix ausgegebenen *p*-Werte für den Vergleich der jeweils in Zeile und Spalte angegebenen Gruppen berücksichtigen bereits die α -Adjustierung, sie können also direkt mit dem gewählten Signifikanzniveau verglichen werden. Setzt man `p.adjust.method="none"`, unterbleibt eine α -Adjustierung, was zusammen mit `pool.sd=FALSE` zu jenen Ergebnissen führt, die man mit separat durchgeführten zweiseitigen *t*-Tests ohne α -Adjustierung erhalten hätte.

Simultane Konfidenzintervalle nach Tukey

Die Konstruktion simultaner Vertrauensintervalle nach Tukey für die paarweisen Differenzen der Gruppenerwartungswerte (Tukey *honestly significant differences*) stellt eine weitere Möglichkeit dar, auf Unterschieden zwischen jeweils zwei Gruppen zu testen.

```
> TukeyHSD(x=aov-Objekt), which="Faktor", conf.level=0.95)
```

Als Argument `x` erwartet `TukeyHSD()` ein von `aov()` erstelltes Objekt. Wurde in ihm mehr als ein Faktor berücksichtigt, kann mit `which` in Form eines Vektors aus Zeichenketten angegeben werden, welche dieser Faktoren für die Bildung von Gruppen herangezogen werden sollen. Über `conf.level` wird die Breite des Vertrauensintervalls für die jeweilige Differenz zweier Erwartungswerte festgelegt.

```
> (tHSD <- TukeyHSD(aovCRp))
Tukey multiple comparisons of means
95% family-wise confidence level
Fit: aov(formula = DV ~ IV, data = dfCRp)
```

	diff	lwr	upr	p adj
B-A	-0.1341170	-0.706534340	0.4383004	0.9292665
C-A	0.4314821	-0.122736070	0.9857003	0.1843991
D-A	0.5836704	0.022649650	1.1446911	0.0380018
C-B	0.5655991	-0.003576610	1.1347748	0.0521308
D-B	0.7177873	0.141985791	1.2935888	0.0079573
D-C	0.1521882	-0.405524534	0.7099010	0.8935478

Die Ausgabe umfasst für jeden Gruppenvergleich die beobachtete Mittelwertsdifferenz, ihr Vertrauensintervall sowie den zugehörigen *p*-Wert. Auch `glht()` aus dem Paket `multcomp` kann alle paarweisen Gruppenvergleiche mit der α -Adjustierung nach Tukey testen, indem das Argument `linfct` auf `mcp(UV)="Tukey"` gesetzt wird.

```
> library(multcomp) # für glht()
> tukey <- glht(aovCRp, linfct=mcp(IV="Tukey")) # Tukey Kontraste
> summary(tukey) # Tests ...
> confint(tukey) # Konfidenzintervalle ...
```

Die Ergebnisse lassen sich manuell nachvollziehen, wobei hier zu beachten ist, dass ungleiche Gruppengrößen vorliegen. Kritischer Wert und Verteilungsfunktion der Teststatistik sind mit `ptukey()` bzw. `ptukey()` zu berechnen.

```
> Mj <- tapply(dfCRp$DV, dfCRp$IV, mean)           # Gruppenmittelwerte
> Nj <- table(dfCRp$IV)                            # Gruppengrößen

# alle paarweisen Mittelwertsdifferenzen
> diffMat <- outer(Mj, Mj, "-")
> (diffs <- diffMat[lower.tri(diffMat)])
[1] -0.1341170 0.4314821 0.5836704 0.5655991 0.7177873 0.1521882

# Skalierungsfaktor: Wurzel aus Summe der Kehrwerte je zweier Nj
> NjMat <- sqrt(outer(1/Nj, 1/Nj, "+"))
> NjFac <- NjMat[lower.tri(NjMat)]
> qTs <- abs(diffs) / (sqrt(MSw/2) * NjFac)      # Teststatistiken
> (pVals <- ptukey(qTs, P, dfSSw, lower.tail=FALSE)) # p-Werte
[1] 0.929266544 0.184399137 0.038001832 0.052130777 0.007957315 0.893547830

# halbe Breite Vertrauensintervall für paarweise Mittelwertsdifferenz
> tWidth <- qtukey(0.05, P, dfSSw, lower.tail=FALSE) * sqrt(MSw/2) * NjFac
> diffs - tWidth                                # Vertrauensintervalle untere Grenzen
[1] -0.706534340 -0.122736070 0.022649650 -0.003576610
[5] 0.141985791 -0.405524534

> diffs + tWidth                                # Vertrauensintervalle obere Grenzen
[1] 0.4383004 0.9857003 1.1446911 1.1347748 1.2935888 0.7099010
```

Die Konfidenzintervalle können grafisch veranschaulicht werden, indem das Ergebnis von `TukeyHSD()` einem Objekt zugewiesen und dies an `plot()` übergeben wird (Abb. 7.2). Die gestrichelt gezeichnete senkrechte Linie markiert die 0 – Intervalle, die die 0 enthalten, entsprechen einem nicht signifikanten Paarvergleich.

```
> plot(tHSD)
```

7.4 Einfaktorielle Varianzanalyse mit abhängigen Gruppen (RB-*p*)

In einem RB-*p* Design werden in *p* Bedingungen einer UV abhängige Beobachtungen gemacht. Diese Situation verallgemeinert jene eines *t*-Tests für abhängige Stichproben auf mehr als zwei Gruppen, d. h. auf jeweils mehr als zwei voneinander abhängige Beobachtungen.

Jede Menge aus *p* abhängigen Beobachtungen (eine aus jeder Bedingung) wird als *Block* bezeichnet. Versuchsplänerisch ist zu beachten, dass im Fall der Messwiederholung die Reihenfolge der Beobachtungen für jede Person randomisiert werden muss, weswegen das Design unter *randomized block design* firmiert. Bei gematchten Personen ist zunächst die Blockbildung so vorzunehmen, dass jeder Block *p* Personen umfasst, die bzgl. relevanter Störvariablen homogen

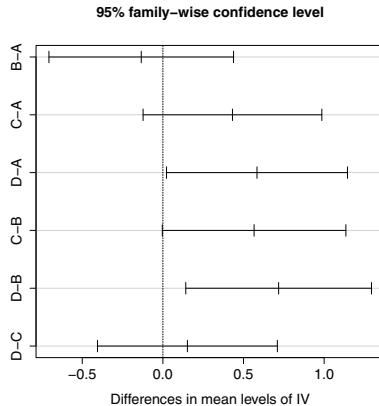


Abbildung 7.2: Grafische Darstellung simultaner Vertrauensintervalle nach Tukey

sind. Innerhalb jedes Blocks müssen daraufhin die Personen randomisiert den Bedingungen zugeordnet werden.

Im Vergleich zum CR-*p* Design wirkt im Modell zum RB-*p* Design ein systematischer Effekt mehr am Zustandekommen einer Beobachtung mit: Zusätzlich zum Effekt der Zugehörigkeit zu einer Bedingung ist dies der Blockeffekt aus der Zugehörigkeit einer Beobachtung zu einem Block. Im Gegensatz zum festen (*fixed*) Gruppenfaktor stellt die Blockzugehörigkeit einen zufälligen (*random*) Faktor dar. Die Bezeichnungen leiten sich daraus ab, dass die Faktorstufen inhaltlich bedeutsam sind, durch den Versuchsleiter reproduzierbar festgelegt werden und eine Generalisierung der Ergebnisse über diese Stufen hinaus nicht erfolgt. Die vorliegenden Ausprägungen der Blockzugehörigkeit (im Fall der Messwiederholung die Personen selbst) sind dagegen selbst nicht bedeutsam. Stattdessen sind sie nicht kontrollierte Realisierungen einer Zufallsvariable im statistischen Sinn. Dies erlaubt es den Ergebnissen, über die tatsächlich realisierten Werte hinaus Gültigkeit beanspruchen können.

7.4.1 Univariat formuliert auswerten und Effektstärke schätzen

Für die Durchführung der Varianzanalyse ergeben sich wichtige Änderungen im Vergleich zum CR-*p* Design. Zunächst sind die statistischen Voraussetzungen andere: Die jeweils pro Block erhobenen Daten müssen, als Zufallsvektoren aufgefasst, gemeinsam normalverteilt sein. Darüber hinaus muss Zirkularität gelten (vgl. Abschn. 7.4.2). Für die Durchführung des Tests mit `aov()` muss der Datensatz so strukturiert sein, dass jede Zeile nicht die Werte eines einzelnen Blocks zu allen Messzeitpunkten beinhaltet (Wide-Format), sondern auch die Messwerte eines Blocks aus verschiedenen Messzeitpunkten in separaten Zeilen stehen (Long-Format, vgl. Abschn. 3.3.9). Der Messzeitpunkt muss mit einem Objekt der Klasse `factor` codiert werden. Weiterhin muss es eine Variable geben, die codiert, von welcher Person, bzw. allgemeiner aus welchem Block ein Wert der AV stammt. Dieser Blockbildungsfaktor sei im folgenden als `Block` bezeichnet

und muss ein Objekt der Klasse `factor` sein.⁸

Weil die Fehlerstruktur der Messwerte blockweise Abhängigkeiten aufweist, ändert sich die Modellformel in den Aufrufen von `aov()` dahingehend, dass explizit anzugeben ist, aus welchen additiven Komponenten sich die Quadratsumme innerhalb der Gruppen zusammensetzt. Im RB-*p* Design drückt `Error(<Block>/<UV>)` aus, dass `<Block>` in `<UV>` verschachtelt ist und der durch die Variation der UV entstehende Effekt in der AV jeweils innerhalb der durch `<Block>` definierten Blöcke analysiert werden muss.⁹

```
> aov(<AV> ~ <UV> + Error(<Block>/<UV>), data=<Datensatz>)

> N      <- 10                                # Zellbesetzung
> P      <- 4                                 # Anzahl UV-Stufen
> id     <- factor(rep(1:N, times=P))        # Blockzugehörigkeit
> IV     <- factor(rep(1:P, each=N))         # Intra-Gruppen Faktor
> DV_t1 <- round(rnorm(N, -0.3, 1), 2)       # AV zu t1
> DV_t2 <- round(rnorm(N, -0.2, 1), 2)       # AV zu t2
> DV_t3 <- round(rnorm(N, 0.1, 1), 2)        # AV zu t3
> DV_t4 <- round(rnorm(N, 0.4, 1), 2)        # AV zu t4
> DV     <- c(DV_t1, DV_t2, DV_t3, DV_t4)    # Gesamt-Daten

> dfRBpL <- data.frame(id, IV, DV)           # Datensatz Long-Format
> aovRBp <- aov(DV ~ IV + Error(id/IV), data=dfRBpL)
> summary(aovRBp)
Error: id
      Df  Sum Sq Mean Sq F value Pr(>F)
Residuals  9 15.115  1.6795

Error: id:IV
      Df  Sum Sq Mean Sq F value Pr(>F)
IV          3 12.569  4.1895  3.9695 0.01823 *
Residuals 27 28.496  1.0554
```

Die beim Test eines Effekts jeweils verwendete Quelle der Fehlervarianz wird durch die `Error: <\rightarrow Effekt>` Überschriften kenntlich gemacht, ihre Quadratsumme findet sich in der Zeile `Residuals`. Es ist zu erkennen, dass im RB-*p* Design die Quadratsumme des festen Effekts (IV) gegen die Quadratsumme der Interaktion von festem und Random-Faktor getestet wird (`Error: \rightarrow id:IV`). Dies wird auch deutlich, wenn die Daten eines RB-*p* Designs mit einer zweifaktoriellen Varianzanalyse im CRF-*pq* Design analysiert werden (vgl. Abschn. 7.5), wobei der Random- und der feste Faktor jeweils die Rolle einer UV einnehmen:

```
> anova(lm(DV ~ id + IV + id:IV, data=dfRBpL))
Analysis of Variance Table
      Df  Sum Sq Mean Sq F value Pr(>F)
```

⁸Bei Verwendung von `reshape()` werden sowohl die Blockzugehörigkeit als auch der Messzeitpunkt als numerischer Vektor codiert. Beide Variablen sind deshalb manuell in Faktoren umzuwandeln.

⁹Ausgeschrieben lautet der Term `Error(<Block> + <Block>:<UV>)`. Dies sind die beiden Effekte, deren Quadratsummen sich zur Quadratsumme innerhalb der Gruppen addieren – also zur Quadratsumme der Residuen einer CR-*p* ANOVA, die keinen Effekt von `<Block>` berücksichtigt (`anova(lm(DV ~ IV, data=dfRBpL))`).

id	9	15.115	1.6795
IV	3	12.569	4.1895
id:IV	27	28.496	1.0554
Residuals	0	0.000	

Die sich in dieser Varianzanalyse ergebende Quadratsumme der Interaktion beider UVn (**id:IV**) ist identisch zu jener der Residuen beim Test des festen Effekts im RB-*p* Design. Analoges gilt für den Effekt der Variable **id**, dessen Quadratsumme gleich der ersten Residual-Quadratsumme im RB-*p* Design ist. Die Effekt-Quadratsumme von **IV** ist in beiden Varianzanalysen notwendigerweise identisch. Da aus jedem Block nur eine Beobachtung pro Stufe von **IV** vorliegt, beträgt im CRF-*pq* Design die Zellbesetzung 1. Deshalb kann es keine Abweichungen zum Zellmittelwert geben (**Residuals** beträgt 0) – der Vorhersage des Modells mit beiden Haupteffekten und Interaktionseffekt.

Manuelle Kontrolle

Die Ergebnisse lassen sich auch manuell prüfen, wobei nach dem Bilden der blockweise zentrierten Daten wie im CR-*p* Design vorgegangen werden kann. Lediglich die Freiheitsgrade der Fehler unterscheiden sich, zudem ist das Gesamtmittel der zentrierten Daten 0.

```
# für Quadratsummen: jeden Wert durch zugehöriges Mittel ersetzen
> MiL    <- ave(DV, id, FUN=mean)           # Block
> MjL    <- ave(DV, IV, FUN=mean)          # Messzeitpunkt
> DVctr  <- DV - MiL                      # blockweise zentrierte AV
> MjCtr  <- tapply(DVctr, IV, mean)        # Messzeitpunkt-Mittel
> SSb    <- sum(N * MjCtr^2)                # Quadratsumme UV
> M      <- mean(DV)                      # Gesamtmittel

# Interaktion Block:UV, alternativ analog zu SSb im CR-p Design
# IDxIV <- DVctr - ave(DVctr, IV, FUN=mean)
> IDxIV <- DV - MiL - MjL + M            # Interaktion
> SSE   <- sum(IDxIV^2)                  # Quadratsumme Residuen
> dfSSb <- P-1                          # Freiheitsgrade UV
> dfSSE <- (N-1) * (P-1)                 # Freiheitsgrade Fehler
> (MSb  <- SSb / dfSSb)                 # mittlere QS UV
[1] 4.189542

> (MSE <- SSE / dfSSE)                  # mittlere QS Residuen
[1] 1.055424

> (Fval <- MSb / MSE)                  # Teststatistik F-Wert
[1] 3.969535

> (pVal <- pf(Fval, dfSSb, dfSSE, lower.tail=FALSE)) # p-Wert
[1] 0.01823255
```

Effektstärke schätzen

Als Maß für die Effektstärke kann das generalisierte η_g^2 herangezogen werden, zu dessen Schätzung $\hat{\eta}_g^2$ die Effekt-Quadratsumme an der Summe von ihr selbst mit allen Residual-Quadratsummen relativiert wird. Die Berechnung – inkl. des hier identischen einfachen $\hat{\eta}^2$ und des partiellen $\hat{\eta}_p^2$ – übernimmt `EtaSq(aov-Objekt)` aus dem Paket `DescTools`. Alternativ dient auch die Intraklassenkorrelation *ICC1* als Maß der Effektstärke in RB-*p* Designs (vgl. Abschn. 10.3.3).

```
> library(DescTools)                                # für EtaSq()
> EtaSq(aovRBp, type=1)
  eta.sq   eta.sq.part  eta.sq.gen
IV  0.22372    0.3060661    0.22372
```

Da von `summary(aov-Modell)` zurückgegebene Objekte eine recht verschachtelte Struktur besitzen, lassen sich die Quadratsummen zur manuellen Kontrolle hier leichter aus dem von `anova(lm-Modell)` erzeugten Datensatz extrahieren.

```
> anRes <- anova(lm(DV ~ IV:id, data=dfRBpL))

# Residual-Quadratsummen
> SSEid   <- anRes["id",      "Sum Sq"]
> SSEIVid <- anRes["IV:id",   "Sum Sq"]
> SSb     <- anRes["IV",      "Sum Sq"]          # Effekt-QS
> (pEtaSq <- SSb / (SSb + SSEIVid))           # partielles eta^2
[1] 0.3060661

> SSEtot  <- SSEid + SSEIVid
> (gEtaSq <- SSb / (SSb + SSEtot))            # generalisiertes eta^2
[1] 0.22372
```

7.4.2 Zirkularität der Kovarianzmatrix prüfen

Die Gültigkeit der dargestellten Varianzanalyse für Daten aus einem RB-*p* Design hängt davon ab, ob neben den anderen Voraussetzungen auch die Annahme von Zirkularität der theoretischen Kovarianzmatrix der AV in den einzelnen Gruppen gilt. Sie ist gegeben, wenn alle aus je zwei unterschiedlichen Variablen gebildeten Differenzvariablen dieselbe theoretische Varianz besitzen.¹⁰ Ein Test auf Zirkularität ist der von Mauchly (vgl. Abschn. 7.4.3, 7.4.4).

Für Situationen ohne gegebene Zirkularität sollen Korrekturformeln verhindern, dass in der Varianzanalyse die tatsächliche Wahrscheinlichkeit eines Fehlers erster Art höher als das nominelle α -Niveau ist. Ist p die Anzahl der Gruppen, besteht die Korrektur in der Multiplikation der Zähler- und Nenner-Freiheitsgrade des getesteten *F*-Bruchs mit einem Skalar ε , für den $\frac{1}{p-1} \leq \varepsilon \leq 1$ gilt. ε ist genau dann 1, wenn die theoretische Kovarianzmatrix zirkulär ist, andernfalls schwankt ε in Abhängigkeit vom „Ausmaß der Abweichung von Zirkularität“. Um

¹⁰Dies ist bei nur zwei Gruppen immer der Fall.

die zugehörige Schätzung $\hat{\epsilon}$ zu berechnen, sind u. a. die Methoden von Box (auch als Greenhouse-Geisser-Korrektur bezeichnet) sowie von Huynh und Feldt anwendbar. Es folgt eine manuelle Berechnung dieser Schätzungen, vgl. Abschn. 7.4.3 für die automatische Berechnung.

```
# Schätzung von epsilon nach Greenhouse + Geisser
> DVmat <- cbind(DV_t1, DV_t2, DV_t3, DV_t4)           # Daten im Wide-Format
> N       <- nrow(DVmat)                                # Anzahl Beobachtungen
> S       <- var(DVmat)                                 # empirische Kovarianzmatrix
> P       <- nrow(S)                                    # Anzahl Variablen (UV-Stufen)
> mdS    <- mean(diag(S))                            # Mittelwert der Diagonalelemente von S
> mS     <- mean(S)                                   # Mittelwert aller Elemente von S
> mSr    <- rowMeans(S)                               # Zeilenmittelwerte von S
> num    <- P^2 * (mdS-mS)^2                           # Zähler
> den    <- (P-1)*(sum(S^2) - 2*P*sum(mSr^2) + P^2*mS^2)      # Nenner
> (epsGG <- num/den)
[1] 0.786984

# Schätzung von epsilon nach Huynh + Feldt
> dfId   <- N-1                                     # df Blockeffekt = df Fehler
> (epsHF <- ((dfId+1)*(P-1)*epsGG - 2) / ((P-1) * (dfId - (P-1)*epsGG)))
[1] 1.084971

# setze epsilon nach Hyhn + Feldt auf 1, wenn es größer ist
> epsHF <- min(1, epsHF)

# Multiplikation der Freiheitsgrade mit epsilon und Vergleich der p-Werte
# Greenhouse + Geisser
> (pEpsGG <- pf(Fval, dfSSb*epsGG, dfSSE*epsGG, lower.tail=FALSE))
[1] 0.02875036

# Huynh+Feldt
> (pEpsHF <- pf(Fval, dfSSb*epsHF, dfSSE*epsHF, lower.tail=FALSE))
[1] 0.01823255
```

Im Beispiel bringt die Korrektur der Freiheitsgrade einen größeren *p*-Wert mit sich, wobei dies nicht in jeder Situation der Fall sein muss. Die Korrektur nach Greenhouse und Geisser gilt in Bereichen oberhalb von 0.9 als etwas zu konservativ. Dies ist nicht der Fall für die Korrektur nach Huynh und Feldt, deren Schätzung aber auch zu Werten größer als 1 führen kann. In diesen Fällen sollte die Schätzung auf 1 gesetzt werden. Soll dagegen konservativ getestet werden, ist die Schätzung $\hat{\epsilon}$ pauschal auf das theoretische Minimum $\frac{1}{p-1}$ zu setzen.

Im vorangehenden Abschnitt wurde bereits die Quadratsumme der Interaktion $\langle\text{Block}\rangle:\langle\text{UV}\rangle$ manuell berechnet, die als Quadratsumme der Residuen beim Test des festen Effekts dient. Hier kann zur Berechnung von $\hat{\epsilon}$ nach Greenhouse und Geisser deshalb auch die etwas einfachere Formel verwendet werden, die auf der Kovarianzmatrix der Residuen basiert.

```
# dem Datensatz Interaktion als neue Variable hinzufügen
> dfRBpL <- cbind(dfRBpL, IDxIV)
```

```
# Residuen als Matrix im Wide-Format extrahieren
> errMat <- data.matrix(unstack(dfRBpL, IDxIV ~ IV))
> Serr   <- cov(errMat)                                # Kovarianzmatrix der Residuen
> (epsGG <- (1/(P-1)) * sum(diag(Serr))^2 / sum(Serr^2))
[1] 0.786984
```

7.4.3 Multivariat formuliert auswerten mit Anova()

`Anova()` aus dem `car` Paket erlaubt die Durchführung von Varianzanalysen mit Messwiederholung, wobei sowohl die bereits beschriebene univariate wie auch eine multivariate Auswertung möglich ist (vgl. Abschn. 12.6.4). Ein Vorteil besteht darin, mit einer leicht nachvollziehbaren Syntax direkt anzugeben, bzgl. welcher Faktoren abhängige Messungen vorliegen. Weiterhin berechnet `Anova()` den Mauchly-Test auf Zirkularität und die Schätzungen $\hat{\epsilon}$ als Maß für die Abweichung der Kovarianzmatrix von Zirkularität mit den Methoden von Greenhouse und Geisser sowie von Huynh und Feldt samt der korrigierten *p*-Werte.

```
> Anova(mod=<lm-Modell>, type=c("II", "III"), idata=<Datensatz>,
+       idesign=<Modellformel>)
```

Unter `mod` ist ein mit `lm()` erstelltes lineares Modell zu übergeben, das bei abhängigen Designs multivariat formuliert werden muss – statt einer einzelnen gibt es also mehrere AVn. In der Modellformel `<AV> ~ <UV>` werden dazu auf der linken Seite der `~` die Daten der AV als Matrix zusammengefügt, wobei jede Spalte die Werte der AV in einer Gruppe (z. B. zu einem Messzeitpunkt) beinhaltet. Auf der rechten Seite der `~` werden die Zwischen-Gruppen Faktoren aufgeführt, wenn es sich um ein gemischtes Design handelt (vgl. Abschn. 7.7). Liegt ein reines Intra-Gruppen Design vor, ist als `UV` nur die Konstante `1` anzugeben. Sind etwa die Werte einer an denselben Personen gemessenen AV zu vier Messzeitpunkten in den Variablen `t1`, `t2`, `t3` und `t4` des Datensatzes `myDf` gespeichert, lautet das Modell `lm(cbind(t1, t2, t3, t4) ~ 1, data=myDf)`.

Mit `type` kann der Quadratsummen-Typ beim Test festgelegt werden (Typ II oder Typ III, vgl. Abschn. 7.5.2), der im mehrfaktoriellen Fall mit ungleichen Zellbesetzungen relevant ist. Die Argumente `idata` und `idesign` dienen der Spezifizierung der Intra-Gruppen Faktoren. Hierfür erwartet `idata` einen Datensatz, der die Struktur der unter `mod` in einer Matrix zusammengefassten Daten der AV beschreibt. Im RB-*p* Design beinhaltet `idata` eine Variable der Klasse `factor`, die die Stufen der Intra-Gruppen UV codiert. In jeder Zeile von `idata` gibt dieser Faktor an, aus welcher Bedingung die zugehörige Spalte der Datenmatrix stammt. Im obigen Design mit vier Messzeitpunkten könnte das Argument damit `idata=data.frame(IV=factor(1:4))` lauten – die erste Spalte der Datenmatrix entspricht der ersten Zeile von `idata`, also der Bedingung 1, usw. Unter `idesign` wird eine Modellformel mit den Intra-Gruppen Vorhersagetermen auf der rechten Seite der `~` angegeben, die linke Seite bleibt leer. Im RB-*p* Design lautet das Argument also `idesign=~IV`.

```
# Datensatz im Wide-Format
> dfRBpW <- data.frame(DV_t1, DV_t2, DV_t3, DV_t4)
```

```
# Zwischen-Gruppen Design (hier kein Zwischen-Gruppen Faktor)
> fitRBp <- lm(cbind(DV_t1, DV_t2, DV_t3, DV_t4) ~ 1, data=dfRBpW)
> inRBp <- data.frame(IV=gl(P, 1)) # Intra-Gruppen Design
> library(car) # für Anova()
> AnovaRBp <- Anova(fitRBp, idata=inRBp, idesign=~IV)
```

Der Test des Modells erfolgt mit `summary()`. Als Besonderheit bei der Anwendung auf ein von `Anova()` erzeugtes Modell kann mit den Argumenten `univariate` und `multivariate` angegeben werden, ob die univariaten und multivariaten Kennwerte berechnet werden sollen.

```
> summary(AnovaRBp, multivariate=FALSE, univariate=TRUE) # ...
```

`Anova()` setzt hier voraus, dass die Anzahl der Freiheitsgrade des Blockeffekts ($n - 1$) mindestens so groß ist wie jene des Intra-Gruppen Effekts ($p - 1$). Es müssen also mindestens so viele Blöcke wie Bedingungen vorhanden sein. Ist dies nicht der Fall, kann für die automatische Berechnung der $\hat{\epsilon}$ -Korrekturen wie im folgenden Abschnitt beschrieben auf `anova()` ausgewichen werden.

7.4.4 Multivariat formuliert auswerten mit `anova()`

Die Auswertung des RB-*p* Designs in multivariater Formulierung ist auch mit der bereits bekannten `anova()` Funktion möglich, die es jedoch anders als `Anova()` erlaubt, dass weniger Blöcke als Gruppen vorliegen. Wie bei `Anova()` muss das erste Argument ein mit `lm()` multivariat formuliertes Modell sein und die AV-Struktur in Form des Datensatzes `idata` angegeben werden. Die Ergebnisse sind identisch zur univariat formulierten Auswertung, wenn das Argument `test="Spherical"` gesetzt wird.¹¹ ϵ wird mit den Methoden von Greenhouse und Geisser sowie von Huynh und Feldt geschätzt und samt der korrigierten *p*-Werte ausgegeben.

Die Angabe des zu testenden Effekts erfolgt mit den Argumenten `M` und `X` in Form eines Modellvergleichs. `M` spezifiziert als rechte Seite einer Modellformel das umfassendere Intra-Gruppen Modell mit dem zu testenden Effekt, `X` analog das eingeschränkte Intra-Gruppen Modell ohne diesen Effekt (vgl. Abschn. 6.3.3, 7.3.3, 12.9.6 sowie Dalgaard, 2007). Die Kennwerte des von `X` zu `M` hinzugenommenen Effekts stehen in der Ausgabe in der Zeile (`Intercept`).

```
> anova(fitRBp, M=~IV, X=~1, idata=inRBp, test="Spherical") # ...
```

Der Mauchly-Test auf Zirkularität wird mit der Funktion `mauchly.test()` durchgeführt, deren Argumente dieselben wie beim obigen Aufruf von `anova()` sind.

```
> mauchly.test(fitRBp, M=~IV, X=~1, idata=inRBp) # ...
```

¹¹Bei der multivariaten Formulierung des Modells wird intern aufgrund der generischen `anova()` Funktion automatisch `anova.mlm()` verwendet, ohne dass dies explizit angegeben werden muss (vgl. Abschnitt 15.2.6). Ohne das Argument `test="Spherical"` wird multivariat getestet (vgl. Abschn. 12.6.4).

7.4.5 Einzelvergleiche und alternative Auswertungsmöglichkeiten

Die Voraussetzungen einer Varianzanalyse im RB-*p* Design sind recht restriktiv: So müssen alle Personen zu denselben Messzeitpunkten beobachtet werden. Personen mit unvollständigen Daten, bei denen Beobachtungen einzelner Messzeitpunkte fehlen, können nicht in die Auswertung einfließen. Weiter ist die Zirkularitäts-Voraussetzung in vielen Situationen unplausibel.

Der letztgenannte Punkt ist insbesondere für beliebige Einzelvergleiche relevant, die im Prinzip nach dem Muster jener in Abschn. 7.3.6 für das CR-*p* Design beschriebenen ebenfalls möglich sind. Dabei ist die mittlere Quadratsumme der Interaktion der UV mit dem Blockeffekt in der Rolle der mittleren Residual-Quadratsumme beim CR-*p* Design zu verwenden. Auch die Anzahl der Fehler-Freiheitsgrade ändert sich entsprechend. Dieses Vorgehen gilt aber als anfällig gegenüber Verletzungen der Zirkularitäts-Voraussetzung. Dabei ist die Korrektur der Freiheitsgrade mit einer Schätzung $\hat{\epsilon}$ kein geeignetes Mittel, um die Gefahr einer erhöhten Wahrscheinlichkeit eines α -Fehlers zu verringern. Eine Alternative für Paarvergleiche zwischen Messzeitpunkten sind *t*-Tests mit `pairwise.t.test(..., paired=TRUE)` (vgl. Abschn. 7.3.6).

Bei verletzter Zirkularität bieten verallgemeinerte Schätzgleichungen oder gemischte Modelle (vgl. Abschn. 6.6.4) eine flexible Alternative, um die Abhängigkeitsstruktur von Daten aus mehrfacher Beobachtung derselben Personen zu berücksichtigen. Für gemischte Modelle können Einzelvergleiche durchgeführt werden, die wieder in `glht()` aus dem Paket `multcomp` implementiert sind (vgl. Abschn. 7.3.6).

Eine multivariate Herangehensweise (vgl. Abschn. 7.4.3, 12.6.4) benötigt keine Zirkularitäts-Voraussetzung, ist aber nur für Situationen geeignet, in denen von allen Personen dieselbe Anzahl von Beobachtungen von denselben Messzeitpunkten vorliegen. Eine weitergehende Darstellung alternativer Strategien liefern die Monographien von Kirk (2013) sowie von Maxwell und Delaney (2004).

7.5 Zweifaktorielle Varianzanalyse (CRF-*pq*)

Bei zwei UVn bestehen verschiedene Möglichkeiten, diese versuchsplanerisch zu Kombinationen von Experimentalbedingungen zu verbinden. Hier sei der Fall betrachtet, dass alle Kombinationen von Bedingungen realisiert werden, es sich also um vollständig gekreuzte Faktoren handelt. Zudem sollen beide UVn Zwischen-Gruppen Faktoren darstellen, jede Person also in nur einer Bedingungskombination beobachtet werden. Die Zuteilung von Personen auf Bedingungen erfolge randomisiert. In diesem Fall liegt ein CRF-*pq* Design (*completely randomized factorial*) mit *p* Stufen der ersten und *q* Stufen der zweiten UV vor. Für die analoge multivariate Varianzanalyse mit mehreren AVn vgl. Abschn. 12.7.2.

Bei allen Varianzanalysen mit mehr als einem Faktor ist es wichtig, ob in jeder experimentellen Bedingung dieselbe Anzahl an Beobachtungen vorliegt. Ist dies nicht der Fall, handelt es sich i. a. um ein unbalanciertes Design, bei dem sich die Gesamt-Quadratsumme nicht mehr eindeutig in die Quadratsummen der einzelnen Effekte als additive Komponenten zerlegen lässt (vgl. Abschn. 7.5.2). Im folgenden sei ein balanciertes Design vorausgesetzt.

7.5.1 Auswertung und Schätzung der Effektstärke

Für das CRF-*pq* Design erlaubt es das Modell der Varianzanalyse, drei Effekte zu testen: den der ersten sowie der zweiten UV und den Interaktionseffekt. Jeder dieser Effekte kann in die Modellformel im Aufruf von `lm()` oder `aov()` als modellierender Term eingehen. Hierbei sei daran erinnert, dass der Interaktionseffekt zweier Variablen in einer Modellformel durch den Doppelpunkt : symbolisiert wird.

```
> aov(<AV> ~ <UV1> + <UV2> + <UV1>:<UV2>, data=<Datensatz>)      # bzw.  
> aov(<AV> ~ <UV1> * <UV2>, data=<Datensatz>)
```

Im Beispiel soll neben den beiden eigentlichen UVn auch ein auf diese Bezug nehmender Faktor in den Datensatz aufgenommen werden: Er ignoriert die zweifaktorielle Struktur und codiert die aus der Kombination beider UVn resultierenden Bedingungen als Ausprägungen einer einzelnen UV i. S. der assoziierten einfaktoriellen Varianzanalyse (vgl. Abschn. 2.6.2, 7.5.4).

```
> Njk <- 8                                # Zellbesetzung  
> P    <- 2                                # Anzahl Stufen UV1  
> Q    <- 3                                # Anzahl Stufen UV2  
> IV1 <- factor(rep(1:P, times=Q*Njk))     # Ausprägungen UV1  
> IV2 <- factor(rep(1:Q, each=P*Njk))       # Ausprägungen UV2  
  
# Kombination zu einem Faktor -> assoziierte einfaktorielle ANOVA  
> IVcomb <- interaction(IV1, IV2)  
  
# UV-Effekte 1.1 2.1 1.2 2.2 1.3 2.3: Haupteffekte, keine Interaktion  
> IVEff   <- c(0.5, -0.5, 0, -1, 1, 0)  
> DV      <- IVEff[unclass(IVcomb)] + rnorm(Njk*P*Q, 0, 1) # Gesamt-AV  
> dfCRFpq <- data.frame(IV1, IV2, IVcomb, DV)      # Datensatz  
> aovCRFpq <- aov(DV ~ IV1*IV2, data=dfCRFpq)  
> summary(aovCRFpq)  
  Df  Sum Sq  Mean Sq  F value    Pr(>F)  
IV1       1  17.650   17.650  14.5870  0.0004349 ***  
IV2       2   1.899    0.949   0.7846  0.4628961  
IV1:IV2   2   3.742    1.871   1.5462  0.2249378  
Residuals 42  50.818    1.210
```

Die varianzanalytische Auswertung eines Designs mit drei vollständig gekreuzten Zwischen-Gruppen Faktoren (CRF-*pqr*) unterscheidet sich nur wenig von der beschriebenen zweifaktoriellen Situation. Lediglich die Modellformel mit den zu berücksichtigenden Effekten ist entsprechend anzupassen. Sollen alle Haupteffekte, Interaktionseffekte erster Ordnung und der Interaktionseffekt zweiter Ordnung eingehen, könnte die Modellformel etwa $\langle AV \rangle \sim \langle UV1 \rangle * \langle UV2 \rangle * \langle UV3 \rangle$ lauten. Um den Interaktionseffekt zweiter Ordnung auszuschließen, wäre die Formulierung $\langle AV \rangle \sim \langle UV1 \rangle * \langle UV2 \rangle * \langle UV3 \rangle - \langle UV1 \rangle : \langle UV2 \rangle : \langle UV3 \rangle$ möglich

Mittelwertsdiagramme

Die einer mehrfaktoriellen Varianzanalyse zugrundeliegende Struktur der Daten lässt sich deskriptiv zum einen in Form von Mittelwerts- bzw. Effekttabellen mit `model.tables()` darstellen, wobei die Zellbesetzungen mit aufgeführt werden. Zum anderen kann die Datenlage grafisch über Mittelwertsdiagramme veranschaulicht werden. Neben der Möglichkeit, dies für die Randmittelwerte mit `plot.design()` zu tun, bietet sich auch `interaction.plot()` für die Zellmittelwerte an.

```
> interaction.plot(x.factor=UV1, trace.factor=UV2, response=AV,
+                   fun=mean, col="Farben", lwd=Liniенstärke)
```

Unter `x.factor` wird jener Gruppierungsfaktor angegeben, dessen Ausprägungen auf der Abszisse abgetragen werden. `trace.factor` kontrolliert, welcher zusätzliche Faktor im Diagramm berücksichtigt wird, seine Stufen werden durch unterschiedliche Linien repräsentiert. Sind die Variablen Teil eines Datensatzes, muss dieser den Variablennamen in der Form `(\u2192Datensatz)$\langle Variable \rangle` vorangestellt werden. Das Argument `response` erwartet die auf der Ordinate abzutragende AV. Soll nicht der Mittelwert, sondern ein anderer Kennwert pro Gruppe berechnet werden, akzeptiert das Argument `fun` auch andere Funktionsnamen als die Voreinstellung `mean`. Über `col` und `lwd` können Farbe und Stärke der Linien kontrolliert werden (Abb. 7.3).

```
> plot.design(DV ~ IV1*IV2, data=dfCRFpq, main="Randmittelwerte")
> interaction.plot(IV1, IV2, DV, main="Mittelwertsverläufe",
+                   col=c("red", "blue", "green"), lwd=2)
```

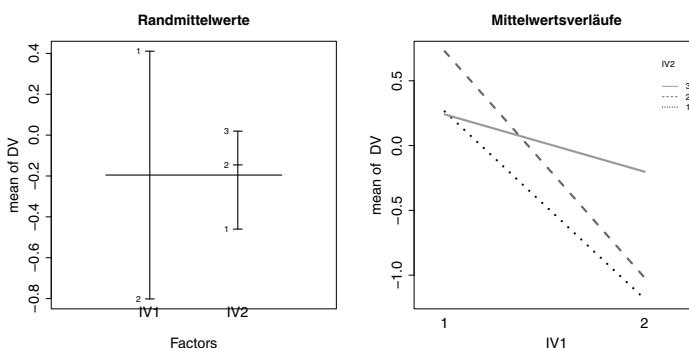


Abbildung 7.3: Darstellung der Randmittelwerte durch `plot.design()` sowie der Gruppenmittelwerte durch `interaction.plot()`

Effektstärke schätzen

Als Maß für die Stärke jedes getesteten Effekts kann das partielle η_p^2 herangezogen werden, zu dessen Schätzung $\hat{\eta}_p^2$ jeweils seine Effekt-Quadratsumme an der Summe von ihr und der

Residual-Quadratsumme relativiert wird. Die Berechnung – inkl. des einfachen $\hat{\eta}^2$ – übernimmt `EtaSq(<aov-Objekt>)` aus dem Paket `DescTools`.

```
> library(DescTools)                                # für EtaSq()
> EtaSq(aovCRFpq, type=1)
    eta.sq   eta.sq.part
IV1      0.23816098  0.25778017
IV2      0.02561867  0.03601418
IV1:IV2  0.05048956  0.06857941

# Kontrolle
> anRes <- anova(lm(DV ~ IV1*IV2, data=dfCRFpq))
> SS1   <- anRes["IV1",       "Sum Sq"]      # Quadratsumme UV1
> SS2   <- anRes["IV2",       "Sum Sq"]      # Quadratsumme UV2
> SSI   <- anRes["IV1:IV2",   "Sum Sq"]      # Quadratsumme Interaktion
> SSE   <- anRes["Residuals", "Sum Sq"]      # Quadratsumme Residuen

> (etaSq1 <- SS1 / (SS1 + SS2 + SSI + SSE))  # eta^2 UV1
[1] 0.238161

> (etaSq2 <- SS2 / (SS1 + SS2 + SSI + SSE))  # eta^2 UV2
[1] 0.02561867

> (etaSqI <- SSI / (SS1 + SS2 + SSI + SSE))  # eta^2 Interaktion
[1] 0.05048956

> (pEtaSq1 <- SS1 / (SS1 + SSE))             # partielle eta^2 UV1
[1] 0.2577802

> (pEtaSq2 <- SS2 / (SS2 + SSE))             # partielle eta^2 UV2
[1] 0.03601418

> (pEtaSqI <- SSI / (SSI + SSE))             # part. eta^2 Interaktion
[1] 0.06857941
```

7.5.2 Quadratsummen vom Typ I, II und III

In mehrfaktoriellen Designs mit unbalancierten Zellbesetzungen offenbart sich, dass R eine andere Methode zur Berechnung der Quadratsummen der Haupteffekte heranzieht, als es der Voreinstellung etwa von SAS oder SPSS entspricht.¹² Für eine ausführlichere Darstellung vgl. Abschn. 12.9.2, 12.9.6 sowie Maxwell und Delaney (2004).

R berechnet sequentielle Quadratsummen vom Typ I, für die bei unbalancierten Zellbesetzungen die Reihenfolge der im Modell berücksichtigten Terme bedeutsam ist. Die Quadratsumme eines

¹²Die hier verwendete Terminologie von *Typen von Quadratsummen* wurde ursprünglich mit dem Programm SAS eingeführt und bezeichnet letztlich unterschiedliche Hypothesen in varianzanalytischen Designs mit mehreren Faktoren.

Effekts wird hier als Reduktion der Quadratsumme der Residuen (*residual sum of squares*, RSS) beim Wechsel zwischen den folgenden beiden Modellen berechnet: dem Modell mit allen Vorhersagetermen, die in der Modellformel vor dem zu testenden Effekt auftauchen und dem Modell, in dem zusätzlich dieser Effekt selbst berücksichtigt wird (vgl. Abschn. 6.3.3). Der resultierende Test ist bei Haupteffekten äquivalent zur Frage, ob die zugehörigen gewichteten Randerwartungswerte identisch sind, wobei die Gewichtung der Zellerwartungswerte bei ihrer Mittelung mit den Zellbesetzungen erfolgt. Die Summe aller Effekt-Quadratsummen ist hier gleich der RSS-Differenz vom vollständigen Modell und jenem, in dem kein Effekt, also nur der Gesamterwartungswert eingeht ($\langle AV \rangle \sim 1$).

Quadratsummen vom Typ II für Haupteffekte berechnen sich als RSS-Differenz beim Wechsel zwischen den zwei folgenden Modellen: dem Modell mit allen Haupteffekten, aber ohne deren Interaktion sowie demselben Modell ohne den jeweils interessierenden Haupteffekt (egal wo er in der Modellformel steht). Die Reihenfolge der Effekte im Modell ist dabei irrelevant. Für den Test der Interaktion wird das vollständige Modell (beide Haupteffekte und Interaktion) mit dem Modell verglichen, das nur beide Haupteffekte berücksichtigt.

Einige andere Programme wie SAS und SPSS dagegen verwenden in der Voreinstellung partielle Quadratsummen vom Typ III. Die Quadratsumme eines Effekts wird hier als RSS-Reduktion beim Wechsel zwischen zwei anderen Modellen berechnet: dem Modell mit allen Vorhersagetermen außer dem interessierenden Effekt (egal ob vor oder nach diesem in der Modellformel stehend) und dem vollständigen Modell mit allen Termen. Die Reihenfolge der Effekte im Modell ist dabei irrelevant. Die Summe aller einzelnen Effekt-Quadratsummen hat hier bei unbalancierten Zellbesetzungen keine Bedeutung. Der so berechnete Test ist bei Haupteffekten äquivalent zur Frage, ob die gleichgewichteten Randerwartungswerte übereinstimmen.¹³

Unabhängig vom Quadratsummen-Typ ist die Fehler-Quadratsumme beim Test einer Hypothese immer jene des vollständigen Modells mit allen Vorhersagetermen der Modellformel. Es handelt sich bei Quadratsummen vom Typ I, II und III letztlich nicht um verschiedene Berechnungsmethoden desselben Kennwertes, stattdessen dienen sie Tests inhaltlich unterschiedlicher Fragestellungen, die sich bei Haupteffekten auf unterschiedliche Modellvergleiche beziehen. Der Test der Interaktion stimmt hingegen bei allen Typen überein. Bei proportional ungleichen Zellbesetzungen¹⁴ ist zudem der Test der Haupteffekte beim Typ I gleich jenem beim Typ II. Im Spezialfall gleicher Zellbesetzungen liefern die Quadratsummen vom Typ I, II und III dieselben Ergebnisse.

Das folgende Beispiel eines CRF-33 Designs mit unbalancierten Zellbesetzungen ist jenes aus Maxwell und Delaney (2004, p. 338 ff.). Zunächst folgen die Modellvergleiche, die zu Quadratsummen vom Typ I führen, daraufhin die Berechnung der Quadratsummen vom Typ III.

```
> mdP <- 3 # Anzahl Stufen UV1
> mdQ <- 3 # Anzahl Stufen UV2
# AV Daten aus Bedingungen 1-1, 1-2, 1-3, 2-1, 2-2, 2-3, 3-1, 3-2, 3-3
```

¹³Die genannte Äquivalenz von Modellvergleichen und Hypothesen über ungewichtete Randerwartungswerte setzt voraus, dass ein passendes Codierschema für kategoriale Variablen verwendet wird, z. B. die Effektcodierung (vgl. Abschn. 12.9.2).

¹⁴Proportional ungleiche Zellbesetzungen liegen vor, wenn $\frac{n_{jk}}{n_{jk'}} = \frac{n_{j'k}}{n_{j'k'}}$ sowie $\frac{n_{jk}}{\bar{n}_{jk'}} = \frac{n_{jk'}}{\bar{n}_{j'k'}}$ für alle j, j', k, k' gilt.

```

> g11 <- c(41, 43, 50)
> g12 <- c(51, 43, 53, 54, 46)
> g13 <- c(45, 55, 56, 60, 58, 62, 62)
> g21 <- c(56, 47, 45, 46, 49)
> g22 <- c(58, 54, 49, 61, 52, 62)
> g23 <- c(59, 55, 68, 63)
> g31 <- c(43, 56, 48, 46, 47)
> g32 <- c(59, 46, 58, 54)
> g33 <- c(55, 69, 63, 56, 62, 67)
> mdDV <- c(g11, g12, g13, g21, g22, g23, g31, g32, g33) # Gesamt-Daten

# Zugehörige Faktoren UV1, UV2, Datensatz
> mdIV1 <- factor(rep(1:mdP, c(3+5+7, 5+6+4, 5+4+6)))
> mdIV2 <- factor(rep(rep(1:mdQ, mdP), c(3,5,7, 5,6,4, 5,4,6)))
> dfMD <- data.frame(IV1=mdIV1, IV2=mdIV2, DV=mdDV)

# Quadratsummen vom Typ I aus anova()
> anova(lm(DV ~ IV1 + IV2 + IV1:IV2, data=dfMD))
Analysis of Variance Table
Response: DV
            Df  Sum Sq  Mean Sq  F value    Pr(>F)
IV1          2   101.11   50.56   1.8102   0.1782
IV2          2  1253.19   626.59  22.4357 4.711e-07 ***
IV1:IV2      4    14.19     3.55   0.1270   0.9717
Residuals   36  1005.42    27.93

# manuelle Berechnung der Quadratsummen vom Typ I aus dem
# Vergleich der sequentiell aufeinander aufbauenden Modelle
> SSI1 <- anova(lm(DV ~ 1,           dfMD), lm(DV ~ IV1,           dfMD))
> SSI2 <- anova(lm(DV ~ IV1,           dfMD), lm(DV ~ IV1+IV2,       dfMD))
> SSIIi <- anova(lm(DV ~ IV1+IV2,   dfMD), lm(DV ~ IV1+IV2+IV1:IV2, dfMD))

> SSI1[2, "Sum of Sq"]                      # QS Typ I von UV1
[1] 101.1111

> SSI2[2, "Sum of Sq"]                      # QS Typ I von UV2
[1] 1253.189

> SSIIi[2, "Sum of Sq"]                     # QS der Interaktion
[1] 14.18714

# Vergleich des Gesamtmodells mit jenem ohne Effekte
> SSIt <- anova(lm(DV ~ 1, dfMD), lm(DV ~ IV1 + IV2 + IV1:IV2, dfMD))
> SSIt[2, "Sum of Sq"]                      # QS Gesamtmodell
[1] 1368.487

# Summe aller einzelnen Quadratsummen vom Typ I

```

```
> SSI1[2, "Sum of Sq"] + SSI2[2, "Sum of Sq"] + SSIi[2, "Sum of Sq"]
[1] 1368.487
```

In R gibt es im wesentlichen zwei Möglichkeiten, Quadratsummen vom Typ III zu erhalten. Zum einen kann mit `drop1()` (vgl. Abschn. 6.3.3) die RSS-Änderung berechnet werden, die sich jeweils ergibt, wenn ein Vorhersageterm aus der Modellformel gestrichen wird und alle anderen beibehalten werden. Diese Vergleiche liegen gerade Quadratsummen vom Typ III zugrunde. Die Quadratsumme der Residuen steht in der Ausgabe in der Zeile `<none>`. Zum anderen testet `Anova()` aus dem `car` Paket mit Quadratsummen vom Typ III, wenn das Argument `type="III"` gesetzt ist (ebenso Quadratsummen vom Typ II, vgl. Abschn. 7.4.3). In beiden Fällen ist das `contrasts` Argument für `lm()` notwendig, um von der Dummy-Codierung (Treatment-Kontraste) zur Effektcodierung der UVn zu wechseln (vgl. Abschn. 12.9.2).

```
# lineares Modell mit Effektcodierung
> fitIII <- lm(DV ~ IV1 + IV2 + IV1:IV2, data=dfMD,
+                 contrasts=list(IV1=contr.sum, IV2=contr.sum))

> drop1(fitIII, ~ ., test="F")                                # QS Typ III aus drop1()
Single term deletions
Model:
DV ~ IV1 + IV2 + IV1:IV2

      Df  Sum of Sq    RSS     AIC   F value   Pr(>F)
<none>        1005.42 157.79
IV1          2     204.76 1210.19 162.13   3.6658   0.03556 *
IV2          2    1181.11 2186.53 188.75  21.1452  8.447e-07 ***
IV1:IV2      4     14.19 1019.61 150.42   0.1270   0.97170

> library(car)                                              # für Anova()
> Anova(fitIII, type="III")                                  # QS Typ III ...
```

Die Quadratsumme der Interaktion unterscheidet sich nicht zwischen Typ I, II und III, die hier alle dieselben Modellvergleiche verwenden: Das eingeschränkte Modell ist immer das mit beiden Haupteffekten, das vollständige bzw. sequentiell folgende Modell jenes mit beiden Haupteffekten und ihrer Interaktion. Es folgt die manuelle Berechnung der Quadratsummen vom Typ III der Haupteffekte.

```
> (MjkMD <- tapply(mdDV, list(mdIV1, mdIV2), mean))    # Zellmittelwerte
      1     2     3
1 44.66667 49.40 56.85714
2 48.60000 56.00 61.25000
3 48.00000 54.25 62.00000

> (NjkMD <- table(mdIV1, mdIV2))                          # Zellbesetzungen
  1 2 3
1 3 5 7
2 5 6 4
3 5 4 6
```

```

> Mj <- rowMeans(MjkMD)                      # ungewichtete Zeilen-M
> Mk <- colMeans(MjkMD)                      # ungewichtete Spalten-M

# effektive Rand-Ns aus harmonischem Mittel der Zellen-Ns
> effNj <- 1 / rowMeans(1/NjkMD)            # harm. Mittel Zeilen-N
> effNk <- 1 / colMeans(1/NjkMD)            # harm. Mittel Spalten-N

# Gesamtmittel aus gewichteten Zeilen-Ms, bzw. gewichteten Spalten-Ms
> gM1 <- sum(effNj*Mj) / sum(effNj)          # aus Zeilen-Ms
> gM2 <- sum(effNk*Mk) / sum(effNk)          # aus Spalten-Ms

# QS Typ III - mdP bzw. mdQ * quadrierte Differenzen der Randmittel
# zum zugehörigen Gesamtmittel, gewichtet mit dem effektiven Rand-N
> (SSIII1 <- mdP * sum(effNj * (Mj-gM1)^2))  # QS Typ III UV1
[1] 204.7617

> (SSIII2 <- mdQ * sum(effNk * (Mk-gM2)^2))  # QS Typ III UV2
[1] 1181.105

# QS Residuen - Summe der quadrierten Differenzen zum Zellenmittel
> (SSE <- sum((mdDV - ave(mdDV, mdIV1, mdIV2, FUN=mean))^2))
[1] 1005.424

```

Der *F*-Bruch eines Effekts ergibt sich als Quotient der Effekt-Quadratsumme dividiert durch ihre Freiheitsgrade und der Quadratsumme der Residuen des vollständigen Modells dividiert durch ihre Freiheitsgrade.

```

> dfSS1 <- mdP - 1                         # Freiheitsgrade QS UV1
> dfSS2 <- mdQ - 1                         # Freiheitsgrade QS UV2
> dfSSE <- sum(NjkMD - 1)      # Freiheitsgrade QS Residuen vollst. Modell
> (Fval1 <- (SSIII1/dfSS1) / (SSE/dfSSE))  # Teststatistik F-Wert UV1
[1] 3.665829

> (Fval2 <- (SSIII2/dfSS2) / (SSE/dfSSE))  # Teststatistik F-Wert UV2
[1] 21.14520

> (pVal1 <- pf(Fval1, dfSS1, dfSSE, lower.tail=FALSE))  # p-Wert UV1
[1] 0.03555901

> (pVal2 <- pf(Fval2, dfSS2, dfSSE, lower.tail=FALSE))  # p-Wert UV2
[1] 8.44678e-07

```

7.5.3 Bedingte Haupteffekte testen

Im folgenden sei wieder das balancierte Design aus Abschn. 7.5.1 betrachtet.

Dem Test der bedingten Haupteffekte (*simple effects*) der ersten UV im CRF-*pq* Design liegt die Frage zugrunde, ob in einer festen Stufe k der zweiten UV alle Zellerwartungswerte μ_{jk} mit $1 \leq j \leq p$ übereinstimmen, insbesondere also gleich dem zugehörigen Randerwartungswert $\mu_{.k}$ sind (Tab. 7.1). Für jede der q Stufen der zweiten UV lässt sich ein solcher bedingter Haupteffekt der ersten UV testen. Analog soll der Test eines bedingten Haupteffekts der zweiten UV in einer festen Stufe j der ersten UV prüfen, ob alle Zellerwartungswerte μ_{jk} mit $1 \leq k \leq q$ übereinstimmen und damit gleich dem zugehörigen Randerwartungswert $\mu_{.j}$ sind. Für die zweite UV können p bedingte Haupteffekte getestet werden.

Tabelle 7.1: CRF-23 Designschema mit Zell- und Randerwartungswerten

	UV2 – 1	UV2 – 2	UV2 – 3	Mittel
UV1 – 1	μ_{11}	μ_{12}	μ_{13}	$\mu_{1.}$
UV1 – 2	μ_{21}	μ_{22}	μ_{23}	$\mu_{2.}$
Mittel	$\mu_{.1}$	$\mu_{.2}$	$\mu_{.3}$	μ

Für den Test bedingter Haupteffekte stellt das Paket **phia** (De Rosario-Martinez, 2013) die Funktion **testInteractions()** bereit.

```
> testInteractions((aov-Modell), fixed="Bedingung",  
+                  across="bedingte Haupteffekte",  
+                  adjustment="alpha-Adjustierung")
```

Zunächst ist ein mit **aov()** angepasstes Modell zu übergeben. Für **fixed** ist der Faktor mit den Stufen zu nennen, für die bedingte Haupteffekte des anderen Faktors berechnet werden sollen. Für den Test der bedingten Haupteffekte der UV1 auf allen Stufen der UV2 wäre also UV2 an **fixed** zu übergeben. **across** definiert den Faktor, für den bedingte Haupteffekte zu berechnen sind. Wie der Gefahr eines erhöhten Fehlers erster Art durch wiederholtes Testen zu begegnen ist, wird in der Literatur uneinheitlich beurteilt (für verschiedene Strategien vgl. Kirk, 2013, p. 381 ff.). Über **adjustment** können unterschiedliche Methoden gewählt werden. **testInteractions()** verfügt über viele weitere Testmöglichkeiten, die in der Hilfe erläutert sind.

```
> library(phia)                                     # für testInteractions()  
> testInteractions(aovCRFpq, fixed="IV2", across="IV1", adjustment="none")  
F Test:  
P-value adjustment method: none  
          Value Df Sum of Sq      F   Pr(>F)  
1        1.44724  1     8.378  6.9243 0.011841 *  
2        1.74857  1    12.230 10.1079 0.002772 **  
3        0.44248  1     0.783  0.6472 0.425629  
Residuals     42   50.818
```

Die Ergebnisse lassen sich manuell prüfen. Für den Test der bedingten Haupteffekte der ersten UV sind dazu zum einen die einfaktoriellen Varianzanalysen mit dieser UV notwendig, die separat für jede Stufe der zweiten UV durchgeführt werden. Die Beschränkung der Daten auf die feste Stufe einer UV lässt sich mit dem **subset** Argument von **aov()** oder **lm()** erreichen,

dem ein geeigneter Indexvektor zu übergeben ist. Die mittlere Effekt-Quadratsumme jeder dieser Varianzanalysen bildet jeweils den Zähler der *F*-Teststatistiken. Wird die Voraussetzung der Varianzhomogenität als gegeben erachtet, ist der Nenner des *F*-Bruchs für alle Tests identisch und besteht aus der mittleren Quadratsumme der Residuen der zweifaktoriellen Varianzanalyse.

```
# einfaktorielle ANOVAs für UV1 in jeder Stufe der UV2
> CRFp1 <- anova(lm(DV ~ IV1, data=dfCRFpq, subset=(IV2==1)))
> CRFp2 <- anova(lm(DV ~ IV1, data=dfCRFpq, subset=(IV2==2)))
> CRFp3 <- anova(lm(DV ~ IV1, data=dfCRFpq, subset=(IV2==3)))

# extrahiere Effekt-Quadratsummen für bedingte Haupteffekte der UV1
> SSp1 <- CRFp1["IV1", "Sum Sq"]           # in Stufe 1 der UV2
> SSp2 <- CRFp2["IV1", "Sum Sq"]           # in Stufe 2 der UV2
> SSp3 <- CRFp3["IV1", "Sum Sq"]           # in Stufe 3 der UV2

# zweifaktorielle ANOVA: QS und df UV1, Interaktion, Residuen
> CRFpq <- anova(lm(DV ~ IV1*IV2, data=dfCRFpq))
> SSA   <- CRFpq["IV1",      "Sum Sq"]    # QS UV1
> SSI   <- CRFpq["IV1:IV2",  "Sum Sq"]    # QS Interaktion
> SSE   <- CRFpq["Residuals", "Sum Sq"]   # QS Residuen
> dfSSA <- CRFpq["IV1",      "Df"]        # Freiheitsgrade QS UV1
> dfSSE <- CRFpq["Residuals", "Df"]       # Freiheitsgrade Fehler

# (Summe der QS der bedingten Effekte der UV1) = (QS UV1 + QS Interaktion)
> all.equal(SSp1 + SSp2 + SSp3, SSA + SSI)
[1] TRUE

# F-Werte der bedingten Haupteffekte der UV1
> Fp1 <- (SSp1/dfSSA) / (SSE/dfSSE)        # in Stufe 1 der UV2
> Fp2 <- (SSp2/dfSSA) / (SSE/dfSSE)        # in Stufe 2 der UV2
> Fp3 <- (SSp3/dfSSA) / (SSE/dfSSE)        # in Stufe 3 der UV2

# zugehörige nicht adjustierte p-Werte der bedingten Haupteffekte
# bedingter Haupteffekt UV1 in Stufe 1 der UV2
> (pP1 <- pf(Fp1, dfSSA, dfSSE, lower.tail=FALSE))
[1] 0.01184106

# bedingter Haupteffekt UV1 in Stufe 2 der UV2
> (pP2 <- pf(Fp2, dfSSA, dfSSE, lower.tail=FALSE))
[1] 0.002771786

# bedingter Haupteffekt UV1 in Stufe 3 der UV2
> (pP3 <- pf(Fp3, dfSSA, dfSSE, lower.tail=FALSE))
[1] 0.4256288
```

7.5.4 Beliebige a-priori Kontraste

Im Vergleich zum einfaktoriellen Fall im CRF-*p* Design ergeben sich für beliebige a-priori Kontraste im CRF-*pq* Design einige Änderungen, da es nun verschiedene Typen von Kontrasten gibt. Zunächst sind dies jene Kontraste, die sich auf die *assoziierte* einfaktorielle Varianzanalyse beziehen. Dies bedeutet, dass die faktorielle Struktur der Bedingungskombinationen ignoriert wird, stattdessen werden alle Bedingungskombinationen als Stufen einer einzigen (künstlichen) UV betrachtet. Aus einem Design mit zwei Stufen der ersten und drei Stufen der zweiten UV würde so eines mit $2 \cdot 3 = 6$ Stufen einer einzigen UV. Innerhalb dieser assoziierten einfaktoriellen Situation lassen sich nun Kontraste wie in Abschn. 7.3.6 formulieren und testen.

Die mittlere Quadratsumme der Residuen aus der zweifaktoriellen Varianzanalyse ist dieselbe wie die mittlere Quadratsumme innerhalb der Gruppen aus der assoziierten einfaktoriellen Varianzanalyse. Zu beachten ist die Reihenfolge der Stufen in der assoziierten einfaktoriellen Situation. Sie wird hier durch die Stufen des künstlichen Faktors bestimmt, dessen Ausprägung von der Kombination der Faktorstufen der beiden UVn abhängt.

Im Beispiel soll das bereits verwendete CRF-23 Design mit dem in Tab. 7.1 dargestellten Schema vorliegen. Als assoziiertes einfaktorielles Schema ergibt sich das in Tab. 7.2 aufgeführte.

Tabelle 7.2: Assoziiertes einfaktorielles Schema zum CRF-23 Design

UVcomb – 1	UVcomb – 2	UVcomb – 3	UVcomb – 4	UVcomb – 5	UVcomb – 6	M
μ_{11}	μ_{21}	μ_{12}	μ_{22}	μ_{13}	μ_{23}	μ

```
# assoziierte einfaktorielle sowie die zweifaktorielle ANOVA
> CRFpq1 <- aova(lm(DV ~ IVcomb, data=dfCRFpq))
> CRFpq2 <- aova(lm(DV ~ IV1*IV2, data=dfCRFpq))

# prüfe MS within = MS error
> all.equal(CRFpq1[["Residuals", "Mean Sq"]], CRFpq2[["Residuals", "Mean Sq"]])
[1] TRUE
```

Mit `glht()` aus dem Paket `multcomp` soll im Beispiel das Mittel der Erwartungswerte μ_{11} und μ_{12} gegen das Mittel der verbleibenden vier Gruppenerwartungswerte getestet werden. Unter H_0 soll der Kontrast gleich 0 sein, unter H_1 größer als 0.

```
> library(multcomp) # für glht()
> aovComb <- aova(DV ~ IVcomb, data=dfCRFpq) # aov-Modell

# Matrix der Kontrastkoeffizienten - hier nur eine Zeile
> cntrMat <- rbind("contr 01"=c(1/2, -1/4, 1/2, -1/4, -1/4, -1/4))
> summary(glht(aovComb, linfct=mcp(IVcomb=cntrMat), alternative="greater"))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aova(formula = DV ~ IVcomb, data = dfCRFpq)
Linear Hypotheses:
Estimate Std. Error t value Pr(>t)
contr 01 <= 0 1.0372 0.3368 3.08 0.00182 **
```

Da die Gruppengrößen hier gleich sind, vereinfacht sich die Formel für die Teststatistik in der manuellen Berechnung, da nicht mehr gruppenweise mit der Anzahl der Beobachtungen gewichtet werden muss.

```
> Mjk      <- tapply(DV, IVcomb, mean)                      # Zellenmittel
> dfSSE    <- (Njk-1)*P*Q                                  # df von QS error
> SSE      <- sum((DV - ave(DV, IVcomb, FUN=mean))^2)     # QS error
> MSE      <- SSE / dfSSE                                 # MS error
> (psiHat <- sum(cntrMat[1, ] * Mjk))                     # Kontrastschätzung
[1] 1.03724

> lenSq   <- sum(cntrMat[1, ]^2 / Njk)                    # quadrierte Länge
> (tStat <- psiHat / sqrt(lenSq*MSE))                      # Teststatistik t
[1] 3.079714

> (tCrit <- qt(0.05, dfSSE, lower.tail=FALSE))           # krit. t-Wert eins.
[1] 1.681952

> (pVal <- pt(abs(tStat), dfSSE, lower.tail=FALSE))       # p-Wert einseitig
[1] 0.001822898

# untere Grenze des einseitigen 95%-Vertrauensintervalls für psi
> (ciLo <- psiHat - tCrit*sqrt(lenSq*MSE))
[1] 0.4707625
```

Sollen mehrere Kontraste gleichzeitig getestet werden, ist dies durch Zusammenstellung der zugehörigen Kontrastvektoren als (ggf. benannte) Zeilen einer Matrix möglich. Hier werden drei Kontraste ohne Adjustierung des α -Niveaus getestet.

```
# Matrix der Kontrastkoeffizienten
> cntrMat <- rbind("contr 01"=c( 1/2, -1/4, 1/2, -1/4, -1/4, -1/4),
+                      "contr 02"=c( 0,     0,     1,     0,    -1,     0),
+                      "contr 03"=c(-1/2, -1/2, 1/4,  1/4,  1/4,  1/4))

> library(multcomp)                                         # für glht()
> (sumRes <- summary(glht(aovComb, linfct=mcp(IVcomb=cntrMat),
+                           alternative="greater"), test=adjusted("none")))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = DV ~ IVcomb, data = dfCRFpq)
Linear Hypotheses:
Estimate Std. Error t value Pr(>t)
contr 01 <= 0     1.0372     0.3368   3.080 0.00182 **
contr 02 <= 0     0.4878     0.5500   0.887 0.19009
contr 03 <= 0     0.3969     0.3368   1.178 0.12264

# manuelle Berechnung
> psiHats <- cntrMat %*% Mjk                                # Kontrastschätzungen
```

```
> lenSqs  <- cntrMat^2 %*% (1/rep(Njk, ncol(cntrMat))) # quadr. Längen
> tStats  <- psiHats / sqrt(lenSqs*MSE)           # Teststatistiken
> pVals   <- pt(abs(tStats), dfSSE, lower.tail=FALSE) # p-Werte eins.
> data.frame(psiHats, tStats, pVals)
      psiHats     tStats      pVals
contr 01  1.0372399  3.0797137  0.001822898
contr 02  0.4877878  0.8869062  0.190090003
contr 03  0.3968684  1.1783590  0.122643406
```

Neben allgemeinen Kontrasten, die sich auf das assoziierte einfaktorielle Design beziehen, gibt es in der zweifaktoriellen Situation drei weitere Kontrasttypen, auch *Familien* von Kontrasten genannt: Linearkombinationen der mittleren Erwartungswerte in den Stufen der ersten UV (*A-Kontraste*), solche der mittleren Erwartungswerte in den Stufen der zweiten UV (*B-Kontraste*) und Interaktions-Kontraste (*I-Kontraste*). Alle drei zeichnen sich dadurch aus, dass sich die Koeffizienten der Linearkombination nicht nur insgesamt zu 0 summieren, sondern zudem weitere Nebenbedingungen erfüllen, wenn sie in das Designschema eingetragen werden:

- *A*-Kontraste werden mit Koeffizienten gebildet, die im Designschema in jeder Zeile konstant sind und sich pro Spalte zu 0 summieren. Im obigen Beispiel mit der entsprechenden Reihenfolge der Zellen würde etwa der Kontrast $(\frac{1}{3}, -\frac{1}{3}, \frac{1}{3}, -\frac{1}{3}, \frac{1}{3}, -\frac{1}{3})$ den mittleren Erwartungswert μ_1 gegen den mittleren Erwartungswert μ_2 testen.
- *B*-Kontraste werden mit Koeffizienten gebildet, die im Designschema in jeder Spalte konstant sind und sich pro Zeile zu 0 summieren. Im obigen Beispiel mit der entsprechenden Reihenfolge der Zellen würde etwa der Kontrast $(\frac{1}{2}, \frac{1}{2}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4})$ den mittleren Erwartungswert μ_1 gegen das Mittel der mittleren Erwartungswerte μ_2 und μ_3 testen.
- *I*-Kontraste werden mit Koeffizienten gebildet, die sich im Designschema sowohl pro Zeile als auch pro Spalte zu 0 summieren. Im obigen Beispiel mit der entsprechenden Reihenfolge der Zellen würde etwa der Kontrast $(\frac{1}{2}, -\frac{1}{2}, -\frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, \frac{1}{4})$ die Differenz der Gruppenerwartungswerte $\mu_{11} - \mu_{21}$ gegen das Mittel der beiden Differenzen der Gruppenerwartungswerte $\mu_{12} - \mu_{22}$ und $\mu_{13} - \mu_{23}$ testen.

A-, *B*- und *I*-Kontraste können genauso innerhalb der assoziierten einfaktoriellen Varianzanalyse getestet werden wie allgemeine Kontraste. Statt mit den Gruppenerwartungswerten können *A*- und *B*-Kontraste zudem auch mit den mittleren Erwartungswerten formuliert werden. In diesem Fall ist die Teststatistik entsprechend anders zu bilden, der kritische Wert ändert sich dagegen nicht.

Im Beispiel sollen die oben genannten *A*- und *B*-Kontraste mit Hilfe der mittleren Erwartungswerte formuliert werden.

```
# A-Kontrast aus Randmittelwerten der UV1
> meansA  <- tapply(dfCRFpq$DV, dfCRFpq$IV1, mean)      # Zeilenmittel
> cntrVecA <- c(1, -1)                                     # Kontrastvektor
> psiHatA  <- sum(cntrVecA * meansA)                      # Kontrastschätzung
> lenSqA   <- sum(cntrVecA^2 / (Q*Njk))                  # quadrierte Länge
> (stataA  <- psiHatA / sqrt(lenSqA*MSE))                # Teststatistik
[1] 3.819294
```

```
# B-Kontrast aus Randmittelwerten der UV2
> meansB <- tapply(dfCRFpq$DV, dfCRFpq$IV2, mean)      # Spaltenmittel
> cntrVecB <- c(1, -1/2, -1/2)                          # Kontrastvektor
> psiHatB <- sum(cntrVecB * meansB)                      # Kontrastschätzung
> lenSqB <- sum(cntrVecB^2 / (P*Njk))                  # quadrierte Länge
> (statB <- psiHatB / sqrt(lenSqB*MSE))                 # Teststatistik
[1] -1.178359
```

7.5.5 Beliebige post-hoc Kontraste nach Scheffé

Beliebige Kontraste können auch im Anschluss an eine signifikante Varianzanalyse getestet werden. Die zweifaktorielle Varianzanalyse prüft beim Test der Effekte (zwei Haupteffekte, ein Interaktionseffekt) implizit simultan alle möglichen zugehörigen Kontraste (A , B oder I) – spezifische Hypothesen liegen also bei ihrer Anwendung nicht vor. Aus diesem Grund muss im Anschluss an eine Varianzanalyse bei Einzeltests eine geeignete α -Adjustierung vorgenommen werden, hier vorgestellt nach der Methode von Scheffé.¹⁵ Sie ist in `ScheffeTest()` aus dem Paket `DescTools` implementiert (vgl. Abschn. 7.3.6).

Hier soll wieder das Mittel der Erwartungswerte μ_{11} und μ_{12} gegen das Mittel der verbleibenden vier Gruppenerwartungswerte gerichtet getestet werden. Da es sich hierbei nicht um einen A , B oder I -Kontrast handelt, soll in der assoziierten einfaktoriellen Situation getestet werden, was zu einer sehr konservativen α -Adjustierung führt.

```
# transponiere cntrMat mit t(), da Koeffizienten in einer Zeile stehen
> aovComb <- aov(DV ~ IVcomb, data=dfCRFpq)    # assozierte 1-fakt. ANOVA
> library(DescTools)                                # für ScheffeTest()
> ScheffeTest(aovComb, which="IVcomb", contrasts=t(cntrMat))
Posthoc multiple comparisons of means : Scheffe Test
95% family-wise confidence level
Fit: aov(formula = DV ~ IVcomb, data = dfCRFpq)
$IVcomb
            diff      lwr.ci      upr.ci      pval
1.1,1.2-2.1,2.2,1.3,2.3  1.0372399  -0.1385869  2.213067  0.1153
1.2-1.3                  0.4877878  -1.4323293  2.407905  0.9766
1.2,2.2,1.3,2.3-1.1,2.1  0.3968684  -0.7789584  1.572695  0.9228
```

Für die manuelle Kontrolle gilt zunächst alles bereits für beliebige a-priori Kontraste Ausgeführt. Lediglich die Wahl des kritischen Wertes weicht ab und ergibt sich zur α -Adjustierung aus einer F -Verteilung. Dieser kritische Wert ist mit dem Quadrat der a-priori t -Teststatistik zu vergleichen – die etwa in der von `summary(glht(...))` zurückgegebenen Liste in der Komponente `test$tstat` steht.

```
> (Fstat <- sumRes$test$tstat^2)                      # quadrierte Teststatistik
contr 01   contr 02   contr 03
9.4846366 0.7866025 1.3885299
```

¹⁵Für weitere vgl. `PostHocTest()` aus dem Paket `DescTools`.

```
# df von SS between aus der assoziierten einfaktoriellen Varianzanalyse
> dfSSba <- P*Q - 1

# kritischer F-Wert für quadrierte Teststatistik
> (Fcrit <- dfSSba * qf(0.05, df1=dfSSba, df2=dfSSE, lower.tail=FALSE))
[1] 12.18846

# p-Wert einseitig
> (pVal <- pf(Fstat=dfSSba, dfSSba, dfSSE, lower.tail=FALSE))
contr 01   contr 02   contr 03
0.1152858 0.9766012 0.9227677
```

Die Wahl des kritischen Wertes erfolgte hier so, dass alle möglichen Kontraste aus der assoziierten einfaktoriellen Varianzanalyse zugelassen sind. Sollen die Kontraste dagegen nur aus einem Unterraum des Kontrastraumes stammen, etwa weil jeweils nur Kontraste aus einer Familie (*A*, *B* oder *I*) von Interesse sind, kann der kritische *F*-Wert wie folgt gewählt werden. In diesem Fall erfolgt eine gleichzeitige α -Adjustierung nur für Kontraste aus der gewählten Familie, woraus ein geringerer kritischer Wert resultiert:

- *A*-Kontraste (*p* Stufen): $(P-1) * qf(1-0.05, P-1, dfSSE)$
- *B*-Kontraste (*q* Stufen): $(Q-1) * qf(1-0.05, Q-1, dfSSE)$
- *I*-Kontraste: $(P-1) * (Q-1) * qf(1-0.05, (P-1)*(Q-1), dfSSE)$

7.5.6 Marginale Paarvergleiche nach Tukey

Alternativ zu beliebigen Kontrasten können im Anschluss an eine Varianzanalyse auch die Randerwartungswerte jeweils eines Faktors mit Tukey-Kontrasten paarweise miteinander verglichen werden (vgl. Abschn. 7.3.6). Dafür ist in `TukeyHSD()` für das Argument `which` der Faktor zu nennen, dessen Stufen verglichen werden sollen. Die so durchgeföhrten Tests ignorieren allerdings eine ggf. im Modell vorhandene Interaktion.

```
# Modell ohne Interaktion
> aovCRF <- aov(DV ~ IV1 + IV2, data=dfCRFpq)
> TukeyHSD(aovCRF, which="IV2")
Tukey multiple comparisons of means
95% family-wise confidence level
Fit: aov(formula = DV ~ IV1 + IV2, data = dfCRFpq)
$IV2
      diff      lwr      upr      p adj
2-1 0.3142383 -0.6406700 1.269147 0.7061067
3-1 0.4794984 -0.4754099 1.434407 0.4490851
3-2 0.1652602 -0.7896482 1.120168 0.9076526
```

Auch mit `glht()` aus dem Paket `multcomp` können Tukey Paarvergleiche der Randerwartungswerte getestet werden (vgl. Abschn. 7.3.6).

```
> library(multcomp) # für glht()
> tukey <- glht(aovCRF, linfct=mcp(IV2="Tukey")) # Tukey Kontraste
> summary(tukey) # Tests ...
> confint(tukey) # Konfidenzintervalle
```

7.6 Zweifaktorielle Varianzanalyse mit zwei Intra-Gruppen Faktoren (RBF-*pq*)

Wird jede Person in jeder der von zwei Faktoren gebildeten Bedingungskombinationen beobachtet, spricht man von einem RBF-*pq* Design (*randomized block factorial*) mit p Stufen der ersten und q Stufen der zweiten UV. Die Reihenfolge, in der die $p \cdot q$ Bedingungen durchlaufen werden, ist für jede Person zu randomisieren. Ein Block aus $p \cdot q$ voneinander abhängigen Beobachtungen (eine aus jeder Bedingung) kann dabei auch von unterschiedlichen, z. B. bzgl. relevanter Störvariablen gematchten Personen stammen. Hierbei ist innerhalb eines Blocks die Zuteilung von Personen zu Bedingungen zu randomisieren.

Im Vergleich zum CRF-*pq* Design wirkt im Modell zum RBF-*pq* Design ein systematischer Effekt mehr am Zustandekommen einer Beobachtung mit: Zusätzlich zu den drei festen Effekten, die auf die Gruppenzugehörigkeit bzgl. der beiden UVn zurückgehen (beide Haupteffekte und der Interaktionseffekt), ist dies der zufällige Blockeffekt.

7.6.1 Univariat formuliert auswerten und Effektstärke schätzen

Wie im RB-*p* Design (vgl. Abschn. 7.4) müssen die Daten im Long-Format (vgl. Abschn. 3.3.9) inkl. eines Blockbildungsfaktors `Block` der Klasse `factor` vorliegen. Nicht alle möglichen Blöcke sind auch experimentell realisiert, sondern nur eine Zufallsauswahl – `Block` ist also ein Random-Faktor. In der Modellformel muss explizit angegeben werden, aus welchen additiven Komponenten sich die Quadratsumme innerhalb der Zellen zusammensetzt. Im RBF-*pq* Design drückt `Error(Block)/(UV1)*(UV2))` aus, dass `Block` in den durch die Kombination von `UV1` und `UV2` entstehenden Bedingungen verschachtelt ist: Die durch die kombinierte Variation von UV1 und UV2 entstehenden Effekte in der AV (beide Haupteffekte und der Interaktionseffekt) sind daher jeweils innerhalb der durch `Block` definierten Blöcke zu analysieren.¹⁶

```
> aov((AV) ~ (UV1)*(UV2) + Error((Block)/(UV1)*(UV2))),
+   data=(Datensatz))

> N    <- 10                                # Anzahl Personen
> P    <- 2                                  # Messzeitpunkte UV1
> Q    <- 3                                  # Messzeitpunkte UV2
> id   <- factor(rep(1:N, times=P*Q))       # Blockzugehörigkeit
```

¹⁶Der Term lautet `Error(Block) + (Block):(UV1) + (Block):(UV2) + (Block):(UV1):(UV2)`, wenn er ausgeschrieben wird. Dies sind die vier Effekte, deren Quadratsummen sich zur Quadratsumme innerhalb der Zellen addieren – also zur Quadratsumme der Residuen einer CRF-*pq* ANOVA, die keinen Effekt von `Block` berücksichtigt (`anova(lm(DV ~ IV1*IV2, data=dfRBFpqL))`).

```

> IV1 <- factor(rep(rep(1:P, each=N), times=Q)) # Intra-Gruppen UV1
> IV2 <- factor(rep(rep(1:Q, each=P*N))) # Intra-Gruppen UV2

# Simulation der AV mit Effekt beider UVn, ohne Interaktion
> DV_t11 <- round(rnorm(N, -0.8, 1), 2) # AV zu t1-1
> DV_t12 <- round(rnorm(N, -0.7, 1), 2) # AV zu t1-2
> DV_t13 <- round(rnorm(N, 0.0, 1), 2) # AV zu t1-3
> DV_t21 <- round(rnorm(N, 0.2, 1), 2) # AV zu t2-1
> DV_t22 <- round(rnorm(N, 0.3, 1), 2) # AV zu t2-2
> DV_t23 <- round(rnorm(N, 1.0, 1), 2) # AV zu t2-3
> DV <- c(DV_t11, DV_t21, DV_t12, DV_t22, DV_t13, DV_t23)

> dfRBFpqL <- data.frame(id, IV1, IV2, DV) # Datensatz long
> aovRBFpq <- aov(DV ~ IV1*IV2 + Error(id/(IV1*IV2)), data=dfRBFpqL)
> summary(aovRBFpq)
Error: id
      Df  Sum Sq  Mean Sq F value Pr(>F)
Residuals  9  3.6306  0.4034

Error: id:IV1
      Df  Sum Sq  Mean Sq F value Pr(>F)
IV1       1 16.865  16.8646  11.209 0.00855 ***
Residuals  9 13.541   1.5045

Error: id:IV2
      Df  Sum Sq  Mean Sq F value Pr(>F)
IV2       2 13.797   6.8987  4.5641 0.02493 *
Residuals 18 27.207   1.5115

Error: id:IV1:IV2
      Df  Sum Sq  Mean Sq F value Pr(>F)
IV1:IV2    2  2.2468   1.1234  0.9608 0.4014
Residuals 18 21.0473   1.1693

```

Die Ausgabe unterscheidet sich von jener im CRF-*pq* Design, da hier nicht mehr dieselbe Residual-Quadratsumme für den Test jeder der drei Effekte herangezogen wird. Die beim Test eines Effekts jeweils verwendete Quelle der Fehlervarianz wird durch die **Error: <Effekt>** Überschriften kenntlich gemacht, ihre Quadratsumme findet sich in der Zeile **Residuals**. Die Quadratsumme des festen Faktors **IV1** wird mit der Quadratsumme aus der Interaktion von **IV1** und dem Random-Faktor **id** in der Rolle der Residual-Quadratsumme verglichen. Analoges gilt für den festen Faktor **IV2**. Die Quadratsumme der Interaktion der beiden festen Faktoren wird entsprechend mit der Quadratsumme der Interaktion zweiter Ordnung vom Random-Faktor und beiden festen Faktoren getestet (vgl. mit der Ausgabe von **anova(lm(DV ~ IV1*IV2*id, data=dfRBFpqL))**).

Die varianzanalytische Auswertung eines dreifaktoriellen Designs mit drei Intra-Gruppen Faktoren (RBF-*pqr*) unterscheidet sich nur wenig von der beschriebenen zweifaktoriellen Si-

tuation. Lediglich die Modellformel mit den zu berücksichtigenden Effekten ist entsprechend anzupassen. Sollen alle Haupteffekte, Interaktionseffekte erster Ordnung und der Interaktionseffekt zweiter Ordnung eingehen, könnte die Modellformel etwa $\langle AV \rangle \sim \langle UV1 \rangle * \langle UV2 \rangle * \langle UV3 \rangle + \text{Error}(\langle Block \rangle) / (\langle UV1 \rangle * \langle UV2 \rangle * \langle UV3 \rangle)$ lauten.

Manuelle Kontrolle

Der Test eines Haupteffekts ist äquivalent zum Test im einfaktoriellen RB-*p* Design der zuvor blockweise über die Stufen der jeweils anderen UV gemittelten Daten. Für den Test der UV1 ergibt sich dabei als neuer Wert jedes Blocks für eine Stufe der UV1 der zugehörige Mittelwert des Blocks über die Stufen der UV2 – der Test der UV2 erfolgt analog. Der Test der Interaktion von UV1 und UV2 ist u.a. äquivalent zum Test, ob die bedingten Haupteffekte der UV1 für jede Stufe der UV2 identisch sind. Da die UV1 hier nur zwei Stufen hat, können ihre bedingten Haupteffekte durch die blockweisen Differenzen zwischen beiden Stufen der UV1 geschätzt werden: Als neuer Wert jedes Blocks für eine Stufe der UV2 ergibt sich die Differenz der zugehörigen Messwerte zwischen den Stufen der UV1. Mit diesen Daten lässt sich hier ein zum Test der Interaktion äquivalenter Test im RB-*p* Design durchführen.

```
> dfG <- dfRBFpqL # kürzerer Name für Datensatz

# Datensatz aus blockweise über Stufen der UV2 gemittelten Daten
> mDf1 <- aggregate(DV ~ id + IV1, data=dfG, FUN=mean)

# Datensatz aus blockweise über Stufen der UV1 gemittelten Daten
> mDf2 <- aggregate(DV ~ id + IV2, data=dfG, FUN=mean)

# hier: Datensatz aus blockweisen Differenzen zwischen Stufen der UV1
> dDfI <- aggregate(DV ~ id + IV2, data=dfG, FUN=diff)

# äquivalent zum Test des Haupteffekts der UV1 im RBF-pq Design
> summary(aov(DV ~ IV1 + Error(id/IV1), data=mDf1)) # ...

# äquivalent zum Test des Haupteffekts der UV2 im RBF-pq Design
> summary(aov(DV ~ IV2 + Error(id/IV2), data=mDf2)) # ...

# hier: äquivalent zum Test der Interaktion UV1:UV2 im RBF-pq Design
> summary(aov(DV ~ IV2 + Error(id/IV2), data=dDfI)) # ...
```

Im allgemeinen Fall wäre der Test der Interaktion wie folgt manuell durchzuführen.

```
# für Quadratsumme Interaktion: Mittel der Zellen, UV1, UV2, gesamt
> Mjk <- with(dfG, tapply(DV, list(IV1, IV2), mean)) # Zellen-M
> Mj <- with(dfG, tapply(DV, IV1, mean)) # UV1-Mittel
> Mk <- with(dfG, tapply(DV, IV2, mean)) # UV2-Mittel
> M <- mean(Mjk) # Gesamtmittel

# Interaktion UV1:UV2
```

```

> IV1xIV2 <- c(sweep(sweep(Mjk, 1, Mj, "-"), 2, Mk, "-")) + M
> (SSI      <- N * sum(IV1xIV2^2))                      # Quadratsumme Interaktion
[1] 2.246813

# QS Residuen: jeden Wert durch zugehöriges Mittel ersetzen
> MjkL <- with(dfG, ave(DV,      IV1, IV2, FUN=mean))
> MjL <- with(dfG, ave(DV, id,    IV1,           FUN=mean))
> MikL <- with(dfG, ave(DV, id,    IV2,           FUN=mean))
> MiL  <- with(dfG, ave(DV, id,           FUN=mean))
> MjL  <- with(dfG, ave(DV,      IV1,           FUN=mean))
> MkL  <- with(dfG, ave(DV,      IV2,           FUN=mean))

# Interaktion Block:UV1:UV2 -> Residuen bei Test Interaktion UV1:UV2
> IDxIV1xIV2 <- dfG$DV - MjkL - MikL + MiL + MjL + MkL - M
> (SSE      <- sum(IDxIV1xIV2^2))                      # Kontrolle: QS Residuen
[1] 21.04732

> dfSSI  <- (P-1) * (Q-1)                                # Freiheitsgrade Interaktion
> dfSSE  <- (P-1) * (Q-1) * (N-1)                          # Freiheitsgrade Residuen
> (FvalI <- (SSI/dfSSI) / (SSE/dfSSE))                  # F-Wert Interaktion
[1] 0.9607551

# p-Wert Interaktion
> (pValI <- pf(FvalI, dfSSI, dfSSE, lower.tail=FALSE))
[1] 0.4013768

```

Effektstärke schätzen

Als Maß für die Stärke jedes getesteten Effekts kann das generalisierte η_g^2 herangezogen werden, zu dessen Schätzung $\hat{\eta}_g^2$ jeweils seine Effekt-Quadratsumme an der Summe von ihr selbst mit allen Residual-Quadratsummen relativiert wird. Die Berechnung – inkl. des einfachen $\hat{\eta}^2$ und des partiellen $\hat{\eta}_p^2$ – übernimmt `EtaSq(aov-Objekt)` aus dem Paket `DescTools`.

```

> library(DescTools)                                     # für EtaSq()
> EtaSq(aovRBFpq, type=1)
            eta.sq   eta.sq.part  eta.sq.gen
IV1      0.17150172  0.55465963  0.20493961
IV2      0.14031114  0.33648396  0.17415900
IV1:IV2  0.02284859  0.09645404  0.03320113

```

Da von `summary(aov-Modell)` zurückgegebene Objekte eine recht verschachtelte Struktur besitzen, lassen sich die Quadratsummen zur manuellen Kontrolle hier leichter aus dem von `anova(lm-Modell)` erzeugten Datensatz extrahieren.

```

> anRes <- anova(lm(DV ~ IV1*IV2*id, data=dfRBFpqL))

> SSEid      <- anRes["id",           "Sum Sq"]

```

```

> SSEIV1id    <- anRes["IV1:id",      "Sum Sq"]
> SSEIV2id    <- anRes["IV2:id",      "Sum Sq"]
> SSEIV1IV2id <- anRes["IV1:IV2:id", "Sum Sq"]
> SSEtot <- SSEid + SSEIV1id + SSEIV2id + SSEIV1IV2id
> SS1        <- anRes["IV1",        "Sum Sq"]
> SS2        <- anRes["IV2",        "Sum Sq"]
> SSI        <- anRes["IV1:IV2",    "Sum Sq"]

> (etaSq1  <- SS1 / (SS1 + SS2 + SSI + SSEtot))      # eta^2 UV1
[1] 0.1715017

> (etaSq2  <- SS2 / (SS1 + SS2 + SSI + SSEtot))      # eta^2 UV2
[1] 0.1403111

> (etaSqI   <- SSI / (SS1 + SS2 + SSI + SSEtot))      # eta^2 Interaktion
[1] 0.02284859

> (pEtaSq1 <- SS1 / (SS1 + SSEIV1id))      # partielle eta^2 UV1
[1] 0.5546596

> (pEtaSq2 <- SS2 / (SS2 + SSEIV2id))      # partielle eta^2 UV2
[1] 0.336484

> (pEtaSqI <- SSI / (SSI + SSEIV1IV2id))  # partielle eta^2 Interaktion
[1] 0.09645404

> (gEtaSq1 <- SS1 / (SS1 + SSEtot))         # generalisiertes eta^2 UV1
[1] 0.2049396

> (gEtaSq2 <- SS2 / (SS2 + SSEtot))         # generalisiertes eta^2 UV2
[1] 0.174159

> (gEtaSqI <- SSI / (SSI + SSEtot))         # general. eta^2 Interaktion
[1] 0.03320113

```

7.6.2 Zirkularität der Kovarianzmatrizen prüfen

Auch bei einer Varianzanalyse für Daten aus einem RBF-*pq* Design muss für jeden Test der drei möglichen Effekte die Voraussetzung der Zirkularität der zugehörigen Kovarianzmatrix erfüllt sein (vgl. Abschn. 7.4.2). Meist erfolgt daher separat für jeden Test eine Korrektur der Freiheitsgrade auf Basis der Schätzung $\hat{\epsilon}$ nach Greenhouse und Geisser bzw. nach Huynh und Feldt.

Bei der Schätzung $\hat{\epsilon}$ für die Tests der Haupteffekte ist zunächst jeweils die oben beschriebene blockweise Mittelung vorzunehmen. Die jeweils resultierende Datenmatrix der gemittelten Werte umfasst im Wide-Format so viele Spalten, wie die zu testende UV Stufen besitzt und

repräsentiert nunmehr Daten aus einem einfaktoriellen RB-*p* Design. Die Voraussetzung der Zirkularität bezieht sich auf ihre theoretische Kovarianzmatrix, mit deren empirischen Pendant deshalb der Korrekturfaktor $\hat{\epsilon}$ berechnet wird. Mit zwei Stufen der UV1 ist hier keine ϵ -Korrektur des Tests der UV1 notwendig, da (2×2) -Kovarianzmatrizen immer zirkulär sind. Auch der Test der Interaktion lässt sich hier wie beschrieben auf Daten eines RB-*p* Designs zurückführen und $\hat{\epsilon}$ auf Basis der Kovarianzmatrix der Residuen ermitteln.

```
# epsilon-Schätzung für Test der UV2: Daten als Matrix im Wide-Format
> DVmat <- with(dfG, tapply(DV, list(id, IV2), mean))
```

Die weiteren Berechnungen würden mit jenen in Abschn. 7.4.2 übereinstimmen.

```
# epsilon-Schätzung für Test der Interaktion UV1:UV2 aus Residuen
> dfG <- cbind(dfG, IDxIV1xIV2) # dem Datensatz Residuen hinzufügen
```

```
# extrahiere Residuen als Matrix im Wide-Format
> errMat <- data.matrix(unstack(dfG, IDxIV1xIV2 ~ IV2))
> Serr <- cov(errMat) # Kovarianzmatrix der Residuen
> (epsGGi <- (1 / (Q-1)) * sum(diag(Serr))^2 / sum(Serr^2))
[1] 0.9934177
```

```
> dfId <- N-1 # df Blockeffekt = df Fehler
> (epsHFi <- ((dfId+1)*(Q-1)*epsGGi - 2) / ((Q-1)*(dfId - ((Q-1)*epsGGi))))
[1] 1.273915
```

7.6.3 Multivariat formuliert auswerten

Für eine Beschreibung von `Anova()` aus dem `car` Paket vgl. Abschn. 7.4.3. Im Vergleich zur RB-*p* Situation ändert sich hier im wesentlichen das Intra-Gruppen Design unter `idata` und `idesign`. Da nun zwei Intra-Gruppen Faktoren vorliegen, muss der unter `idata` anzugebende Datensatz zwei Variablen der Klasse `factor` beinhalten. In jeder Zeile von `idata` geben diese Faktoren an, aus welcher Bedingungskombination der beiden UVn die zugehörige Spalte der Datenmatrix im Wide-Format stammt. Unter `idesign` ist es nun möglich, als Vorhersageterme in der Modellformel beide Haupteffekte und deren Interaktion einzutragen.

```
# Datensatz im Wide-Format
> dfRBFpqW <- data.frame(DV_t11, DV_t21, DV_t12, DV_t22, DV_t13, DV_t23)

# Zwischen-Gruppen Design - hier keine Zwischen-Gruppen UV
> fitRBFpq <- lm(cbind(DV_t11, DV_t21, DV_t12, DV_t22, DV_t13, DV_t23) ~ 1,
+ data=dfRBFpqW)

# Intra-Gruppen Design: Aufbau der Datenmatrix in fitRBFpq
> (inRBFpq <- expand.grid(IV1=gl(P, 1), IV2=gl(Q, 1)))
  IV1  IV2
1    1    1
2    2    1
```

```

3   1   2
4   2   2
5   1   3
6   2   3

> library(car)                                # für Anova()
> AnovaRBFpq <- Anova(fitRBFpq, idata=inRBFpq, idesign=~IV1*IV2)
> summary(AnovaRBFpq, multivariate=FALSE, univariate=TRUE)      # ...

```

`Anova()` setzt hier voraus, dass die Anzahl der Freiheitsgrade des Blockeffekts ($(n - 1)$) mindestens so groß ist wie jene der Interaktion beider Intra-Gruppen Faktoren ($(p - 1) \cdot (q - 1)$). Es müssen also mindestens $(p - 1) \cdot (q - 1) + 1$ viele Blöcke vorhanden sein. Ist dies nicht der Fall, kann für die automatische Berechnung der $\hat{\varepsilon}$ -Korrekturen auf `anova()` ausgewichen werden (vgl. Abschn. 7.4.4).

Für jeden Test ist mit den Argumenten `M` und `X` das passende Paar vom umfassenderen und eingeschränkten Intra-Gruppen Modell als rechte Seite einer Modellformel zu formulieren. Hier soll dabei wie bei Quadratsummen vom Typ I sequentiell vorgegangen werden (vgl. Abschn. 7.5.2, 12.9.6). Die Kennwerte des von `X` zu `M` hinzugenommenen Effekts stehen in der Ausgabe in der Zeile (`Intercept`). Für `mauchly.test()` vgl. Abschn. 7.4.4.

```

> anova(fitRBFpq, M=~IV1,                      # Test IV1 ...
+           X=-1,          idata=inRBFpq, test="Spherical")

> anova(fitRBFpq, M=~IV1 + IV2,                 # Test IV2 ...
+           X=-IV1,         idata=inRBFpq, test="Spherical")

> anova(fitRBFpq, M=~IV1 + IV2 + IV1:IV2,       # Test IV1:IV2 ...
+           X=-IV1 + IV2, idata=inRBFpq, test="Spherical")

# Mauchly-Tests auf Zirkularität (P=2 -> IV1 hier nicht notwendig)
> mauchly.test(fitRBFpq, M=~IV1 + IV2,           # IV2 ...
+               X=~IV1,          idata=inRBFpq)

> mauchly.test(fitRBFpq, M=~IV1 + IV2 + IV1:IV2, # IV1:IV2 ...
+               X=~IV1 + IV2, idata=inRBFpq)

```

7.6.4 Einzelvergleiche (Kontraste)

Prinzipiell ist es möglich, auch für ein RBF-*pq* Design beliebige Einzelvergleiche als Tests von Linearkombinationen von Erwartungswerten durchzuführen. Dabei ist die Situation analog zu jener im CRF-*pq* Design, wie sie in Abschn. 7.5.4 beschrieben wurde. Auch hier wären zunächst Vergleiche der mittleren Erwartungswerte des ersten Faktors (*A*-Kontraste) bzw. des zweiten Faktors (*B*-Kontraste) von Interaktionskontrasten (*I*-Kontraste) und allgemeinen Zellvergleichen zu unterscheiden.

Im Vergleich zum CRF- pq Design ergeben sich jedoch Unterschiede bei der jeweils zu verwendenden Quadratsumme der Residuen: Während diese im CRF- pq Design für jeden Test dieselbe ist, wird im RBF- pq Design jeder Test mit einer anderen Residual-Quadratsumme durchgeführt. Ein A-Kontrast wäre im RBF- pq Design gegen die Quadratsumme der Interaktion des Blocks mit der UV1 zu testen, ein B-Kontrast entsprechend gegen die Quadratsumme der Interaktion des Blocks mit der UV2, ein I-Kontrast gegen die Quadratsumme der Interaktion zweiter Ordnung des Blocks mit der UV1 mit der UV2. Die Anzahl der Fehler-Freiheitsgrade ändert sich entsprechend der verwendeten Quadratsumme. Für Vergleiche einzelner Zellen müsste das assoziierte einfaktorielle RB- p Design herangezogen werden. Auch hier stellt sich jedoch die Frage, ob Einzelvergleiche in abhängigen Designs auf diese Weise durchgeführt werden sollten. (vgl. Abschn. 7.4.5).

7.7 Zweifaktorielle Varianzanalyse mit Split-Plot-Design (SPF- $p \cdot q$)

Ein Split-Plot-Design liegt im zweifaktoriellen Fall vor, wenn die Bedingungen eines Zwischen-Gruppen Faktors mit jenen eines Faktors kombiniert werden, bzgl. dessen Stufen abhängige Beobachtungen resultieren – etwa weil es sich um einen Messwiederholungsfaktor handelt. Hat der Zwischen-Gruppen Faktor UV1 p und der Intra-Gruppen Faktor UV2 q Stufen, wobei jede mögliche Stufenkombination auch realisiert wird, spricht man von einem SPF- $p \cdot q$ Design (*split plot factorial*).

7.7.1 Univariat formuliert auswerten und Effektstärke schätzen

Versuchsplanerisch müssen zunächst Blöcke aus jeweils q Beobachtungen gebildet werden. Die Anzahl der Blöcke muss dabei ein ganzzahliges Vielfaches von p sein, um gleiche Zellbesetzungen zu erhalten. Im zweiten Schritt werden die einzelnen Blöcke randomisiert auf die Stufen der UV1 verteilt, wobei sich letztlich in jeder der p Stufen dieselbe Anzahl von Blöcken befinden sollte. Als weiterer Schritt der Randomisierung wird im Fall der Messwiederholung innerhalb jedes Blocks die Reihenfolge der Beobachtungen bzgl. der q Stufen der UV2 randomisiert. Ein Block kann sich auch aus Beobachtungen unterschiedlicher, z. B. bzgl. relevanter Störvariablen homogener Personen zusammensetzen. In diesem Fall ist die Zuordnung der q Personen pro Block zu den q Stufen der UV2 zu randomisieren. UV1 und die Blockzugehörigkeit sind im SPF- $p \cdot q$ Design konfundiert, da jeder Block nur Beobachtungen aus einer Stufe der UV1 enthält.

Wie beim RBF- pq Design (vgl. Abschn. 7.6) müssen die Daten im Long-Format (vgl. Abschn. 3.3.9) inkl. eines Blockbildungsfaktors `(Block)` der Klasse `factor` vorliegen. Nicht alle möglichen Blöcke sind auch experimentell realisiert, sondern nur eine Zufallsauswahl – `(Block)` ist also ein Random-Faktor. In der Modellformel muss explizit angegeben werden, aus welchen additiven Komponenten sich die Quadratsumme innerhalb der Zellen zusammensetzt. Im SPF- $p \cdot q$ Design drückt `Error((Block)/(UV2))` aus, dass `(Block)` in `(UV2)` verschachtelt ist: Der Effekt der UV2 ist daher jeweils innerhalb der durch `(Block)` definierten Blöcke zu analysieren.¹⁷

¹⁷Ausgeschrieben lautet der Term `Error((Block) + (Block):(UV2))`. Dies sind die beiden Effekte, deren Quadratsummen sich zur Quadratsumme innerhalb der Zellen addieren – also zur Quadratsumme der Residuen einer CRF- pq ANOVA, die keinen Effekt von `(Block)` berücksichtigt: `anova(lm(DV ~ IVbtw*IVwth, \`

```

> aov(<AV> ~ <UV1>*<UV2> + Error(<Block>/<UV2>), data=<Datensatz>)

> Nj      <- 10                                # Gruppengröße
> P       <- 3                                 # Stufen Zwischen-UV
> Q       <- 3                                 # Zeitpunkte Intra-UV
> id      <- factor(rep(1:(P*Nj), times=Q))    # Blockzugehörigkeit
> IVbtw  <- factor(rep(LETTERS[1:P], times=Q*Nj)) # Zwischen-UV (between)
> IVwth  <- factor(rep(1:Q, each=P*Nj))        # Intra-UV (within)

# Simulation ohne Effekt IVbtw, mit Effekt IVwth, ohne Interaktion
> DV_t1 <- round(rnorm(P*Nj, -0.5, 1), 2)      # AV zu t1
> DV_t2 <- round(rnorm(P*Nj,  0,  1), 2)      # AV zu t2
> DV_t3 <- round(rnorm(P*Nj,  0.5, 1), 2)     # AV zu t3
> DV     <- c(DV_t1, DV_t2, DV_t3)            # Gesamt-Daten

> dfSPFpqL <- data.frame(id, IVbtw, IVwth, DV)   # Datensatz Long-Format
> summary(aov(DV ~ IVbtw*IVwth + Error(id/IVwth), dfSPFpqL))
Error: id
      Df  Sum Sq Mean Sq F value Pr(>F)
IVbtw     2  1.2424  0.6212  0.5533  0.5814
Residuals 27 30.3135  1.1227

Error: id:IVwth
      Df  Sum Sq Mean Sq F value Pr(>F)
IVwth      2  9.494  4.7470  6.2443 0.003635 **
IVbtw:IVwth 4  4.830  1.2074  1.5882 0.190704
Residuals  54 41.052  0.7602

```

Die beim Test eines Effekts jeweils verwendete Quelle der Fehlervarianz wird durch die `Error: <\rightarrow Effekt>` Überschriften kenntlich gemacht, ihre Quadratsumme findet sich in der Zeile `Residuals`. Die Ausgabe macht deutlich, dass die Quadratsumme des Effekts des Zwischen-Gruppen Faktors `IVbtw` mit der Quadratsumme des Random-Faktors `id` in der Rolle der Residual-Quadratsumme verglichen wird. Sowohl die Quadratsumme des Intra-Gruppen Faktors `IVwth` als auch die der Interaktion beider UVn wird mit der Quadratsumme aus der Interaktion von `IVwth` und `id` in der Rolle der Residual-Quadratsumme verglichen (vgl. mit der Ausgabe von `anova(lm(DV ~ \rightarrow IVbtw*IVwth*id, data=dfSPFpqL))`).

Die in Abschn. 7.7.3 verwendete `Anova()` Funktion bietet die Möglichkeit, bei ungleichen Zellbesetzungen bzgl. des Zwischen-Gruppen Faktors Quadratsummen vom Typ II und III zu berechnen, während `aov()` nur Quadratsummen vom Typ I ermittelt (vgl. Abschn. 7.5.2).

Effektstärke schätzen

Als Maß für die Stärke jedes getesteten Effekts kann das generalisierte η_g^2 herangezogen werden, zu dessen Schätzung $\hat{\eta}_g^2$ jeweils seine Effekt-Quadratsumme an der Summe von ihr selbst mit

`\rightarrow data=dfSPFpqL)).`

allen Residual-Quadratsummen relativiert wird. Die Berechnung – inkl. des einfachen $\hat{\eta}^2$ und des partiellen $\hat{\eta}_p^2$ – übernimmt `EtaSq(aov-Objekt)` aus dem Paket `DescTools`.

```
> library(DescTools)                                # für EtaSq()
> EtaSq(aovSPFpq, type=1)
    eta.sq  eta.sq.part  eta.sq.gen
IVbtw      0.01429186  0.03937160  0.01711126
IVwth      0.10921277  0.18782992  0.11741384
IVbtw:IVwth 0.05555568  0.10526138  0.06338389
```

Da von `summary(aov-Modell)` zurückgegebene Objekte eine recht verschachtelte Struktur besitzen, lassen sich die Quadratsummen zur manuellen Kontrolle hier leichter aus dem von `anova(lm-Modell)` erzeugten Datensatz extrahieren.

```
> anRes <- anova(lm(DV ~ IVbtw*IVwth*id, data=dfSPFpqL))

# Residual-Quadratsummen
> SSEid     <- anRes["id",          "Sum Sq"]
> SSEIVwid <- anRes["IVwth:id",   "Sum Sq"]
> SSEtot    <- SSEid + SSEIVwid
> SSbtw    <- anRes["IVbtw",       "Sum Sq"]  # QS Zwischen-UV
> SSwth    <- anRes["IVwth",       "Sum Sq"]  # QS Intra-UV
> SSI      <- anRes["IVbtw:IVwth", "Sum Sq"]  # QS Interaktion

> (etaSqB  <- SSbtw / (SSbtw + SSwth + SSI + SSEtot)) # eta^2 Zwischen-UV
[1] 0.01429186

> (etaSqW  <- SSbtw / (SSbtw + SSwth + SSI + SSEtot)) # eta^2 Intra-UV
[1] 0.01429186

> (etaSqI  <- SSbtw / (SSbtw + SSwth + SSI + SSEtot)) # eta^2 Interaktion
[1] 0.01429186

> (pEtaSqB <- SSbtw / (SSbtw + SSEid))        # partielle eta^2 Zwischen-UV
[1] 0.0393716

> (pEtaSqW <- SSwth / (SSwth + SSEIVwid))      # partielle eta^2 Intra-UV
[1] 0.1878299

> (pEtaSqI <- SSI    / (SSI    + SSEIVwid))      # partielle eta^2 Interaktion
[1] 0.1052614

> (gEtaSqB <- SSbtw / (SSbtw + SSEtot))         # general. eta^2 Zwischen-UV
[1] 0.01711126

> (gEtaSqW <- SSwth / (SSwth + SSEtot))         # general. eta^2 Intra-UV
[1] 0.1174138
```

```
> (gEtaSqI <- SSI / (SSI + SSEtot))           # general. eta^2 Interaktion
[1] 0.06338389
```

7.7.2 Voraussetzungen und Prüfen der Zirkularität

Die statistischen Voraussetzungen im SPF- $p \cdot q$ Design unterscheiden sich z. T. von jenen in der RBF- pq Situation (für Details vgl. Kirk, 2013, p. 524 ff.). Für den Test des Haupteffekts des Zwischen-Gruppen Faktors UV1 besteht u.a. die Bedingung der Varianzhomogenität. Sie bezieht sich auf die Daten, die innerhalb jeder Stufe der UV1 durch blockweises Mitteln der Werte über die Stufen der UV2 entstehen. Die so gebildeten Mittelwerte müssen in jeder Stufe der UV1 dieselbe theoretische Varianz besitzen. Es gelten also alle Voraussetzungen wie für eine Varianzanalyse im zugehörigen CR- p Design, die zum Test des Haupteffekts der UV1 im SPF- $p \cdot q$ Design äquivalent ist.

```
# Datensatz aus blockweise gemittelten Werten
> mDf <- aggregate(DV ~ id + IVbtw, data=dfSPFpqL, FUN=mean)

# einfaktorielle ANOVA -> äquivalent zum Test der UV1 im SPF-p.q Design
> summary(aov(DV ~ IVbtw, data=mDf))          # ...
```

Für den Test des Haupteffekts des Intra-Gruppen Faktors UV2 und der Interaktion von UV1 und UV2 gelten folgende Voraussetzungen: Die für jede Stufe der UV1 gebildeten theoretischen Kovarianzmatrizen der UV2 müssen identisch und zudem zirkulär sein (vgl. Abschn. 7.4.2). Da im Gegensatz zum RBF- pq Design beide Tests auf derselben Residual-Quadratsumme basieren, gibt es nur eine für beide Tests gültige Korrektur der Freiheitsgrade auf Basis der Schätzung $\hat{\epsilon}$ nach Greenhouse und Geisser bzw. nach Huynh und Feldt.

Für die manuelle Berechnung von $\hat{\epsilon}$ ist es hier notwendig, zunächst explizit die Interaktion des Blockeffekts mit dem Zwischen-Gruppen Faktor zu berechnen, da sie die Residual-Quadratsumme in den Tests des Intra-Gruppen Faktors und der Interaktion von Zwischen- und Intra-Gruppen Faktor im Nenner des *F*-Bruchs liefert.

```
# ersetze AV durch zugehörige Mittelwerte der Zellen, Blöcke, UV1-Stufen
> Mjk <- with(dfSPFpqL, ave(DV, IVbtw, IVwth, FUN=mean))
> Mi  <- with(dfSPFpqL, ave(DV, id,           FUN=mean))
> Mj  <- with(dfSPFpqL, ave(DV, IVbtw,           FUN=mean))

# Interaktion Block:UV2 - Residuen bei Test von UV2, Interaktion UV1:UV2
> IDxIV <- dfSPFpqL$DV - Mi - Mjk + Mj
> sum(IDxIV^2)                                # Kontrolle: QS Residuen
[1] 41.05164

# Interaktion zu dfSPFpqL hinzufügen, als Matrix im Wide-Format extrahieren
> dfSPFpqL <- cbind(dfSPFpqL, IDxIV)
> errMat   <- data.matrix(unstack(dfSPFpqL, IDxIV ~ IVwth))

# Schätzung von epsilon nach Greenhouse + Geisser auf Basis der Residuen
```

```
> Serr <- cov(errMat) # Kovarianzmatrix der Residuen
> (epsGG <- (1 / (Q-1)) * sum(diag(Serr))^2 / sum(Serr^2))
[1] 0.9125422

# Schätzung von epsilon nach Huynh + Feldt
> dfId <- (Nj-1) * P # df Blockeffekt = df Fehler
> (epsHF <- ((dfId+1)*(Q-1)*epsGG - 2) / ((Q-1) * (dfId - (Q-1)*epsGG)))
[1] 0.975224
```

7.7.3 Multivariat formuliert auswerten

Für eine Beschreibung von Anova() aus dem car Paket vgl. Abschn. 7.4.3. Als wesentlicher Unterschied zur RB- p Situation ergibt sich im SPF- $p \cdot q$ Design die Änderung des mit lm() multivariat spezifizierten Zwischen-Gruppen Designs. In der Modellformel ist nun auf der rechten Seite der ~ der Zwischen-Gruppen Faktor zu nennen.

```
# Datensatz im Wide-Format
> IVbtW <- factor(rep(LETTERS[1:P], Nj))
> dfSPFpqW <- data.frame(IVbtW, DV_t1, DV_t2, DV_t3)

# Zwischen-Gruppen Design
> fitSPFpq <- lm(cbind(DV_t1, DV_t2, DV_t3) ~ IVbtW, data=dfSPFpqW)
> intraSPFpq <- data.frame(IVwth=gl(Q, 1)) # Intra-Gruppen Design
> library(car) # für Anova()
> AnovaSPFpq <- Anova(fitSPFpq, idata=intraSPFpq, idesign=~IVwth)
> summary(AnovaSPFpq, multivariate=FALSE, univariate=TRUE) # ...
```

Anova() setzt hier voraus, dass die Anzahl der Freiheitsgrade des Blockeffekts $((n_j - 1) \cdot p)$ bei gleichen Gruppengrößen n_j mindestens so groß wie die des Intra-Gruppen Effekts $(q - 1)$ ist. Es müssen also mindestens $\frac{q-1}{p} + 1$ viele Blöcke pro Gruppe vorhanden sein. Ist dies nicht der Fall, kann für die automatische Berechnung der ε -Korrekturen auf anova() ausgewichen werden (vgl. Abschn. 7.4.4).

Hier ist zu beachten, dass sich die mit X und M definierten Modelle nur auf die Intra-Gruppen Effekte beziehen. Taucht ein Zwischen-Gruppen Faktor in der Formel von lm() auf, testet anova() seine Interaktion mit dem von X zu M hinzugenommenen Intra-Gruppen Effekt, wobei die Quadratsumme des umfassenderen Intra-Gruppen Modells die Rolle der Residual-Quadratsumme einnimmt.

Die Quadratsumme des Zwischen-Gruppen Effekts wird im SPF- $p \cdot q$ Design gegen jene des Blockeffekts getestet. Das mit M definierte umfassendere Intra-Gruppen Modell muss also jenes mit nur diesem Effekt sein (~ 1). Mit X ist entsprechend das eingeschränkte Modell ohne jeden Effekt zu definieren (~ 0). Da von X zu M kein Intra-Gruppen Effekt hinzu kommt, bleibt es beim Test der Quadratsumme des Zwischen-Gruppen Faktors gegen jene des umfassenderen Intra-Gruppen Modells.

```
# Test des Zwischen-Gruppen Faktors IVbtW ...
> anova(fitSPFpq, M=~1, X=~0, idata=intraSPFpq, test="Spherical")
```

Der Intra-Gruppen Faktor lässt sich testen, indem er beim Modellwechsel von X zu M hinzugenommen wird. Die Kennwerte dieses Effekts stehen in der Ausgabe in der Zeile (`Intercept`). Zusätzlich testet `anova()` wie erwähnt die Quadratsumme der Interaktion des Zwischen-Gruppen Faktors mit diesem hinzugenommenen Effekt gegen jene des vollständigen Intra-Gruppen-Modells. Für `mauchly.test()` vgl. Abschn. 7.4.4.

```
# Test IVwth (Zeile Intercept), Interaktion IVbtw:IVwth (Zeile IVbtw) ...
> anova(fitSPFpq, M=-IVwth, X=~1, idata=intraSPFpq, test="Spherical")
> mauchly.test(fitSPFpq, M=-IVwth, X=~1, idata=inSPFpq) # ...
```

7.7.4 Einzelvergleiche (Kontraste)

Im SPF- $p \cdot q$ Design lassen sich beliebige Einzelvergleiche als Tests von Linearkombinationen von Erwartungswerten analog zu jenen im CRF- pq Design testen (vgl. Abschn. 7.5.4). Auch hier sind Vergleiche der mittleren Erwartungswerte des Zwischen-Gruppen Faktors (A -Kontraste) von solchen des Intra-Gruppen Faktors (B -Kontraste) und von Interaktionskontrasten (I -Kontraste) zu unterscheiden.

Im Vergleich zum CRF- pq Design ergeben sich jedoch Unterschiede bei der jeweils zu verwendenden Quadratsumme der Residuen, die dort immer dieselbe ist. Die Effekt-Quadratsumme eines A -Kontrasts ist im SPF- $p \cdot q$ Design gegen die Quadratsumme des Blockeffekts zu testen. Alternativ lassen sich A -Kontraste wie Kontraste im CR- p Design testen, nachdem zu blockweise gemittelten Daten übergegangen wurde. B - sowie Interaktionskontraste sind gegen die Quadratsumme der Interaktion vom Blockeffekt und Intra-Gruppen Faktor mit den zugehörigen Freiheitsgraden $(n_j - p) \cdot (q - 1)$ zu testen (mit gleichen Gruppengrößen n_j). Allerdings stellt sich hier die Frage, ob sich auf Stufen des Intra-Gruppen Faktors beziehende Einzelvergleiche sinnvollerweise durchgeführt werden sollten (vgl. Abschn. 7.4.5).

Im Beispiel soll der A -Kontrast $-0.5(\mu_1 + \mu_2) + \mu_3$ im zugehörigen CR- p Design nach blockweiser Mittelung gerichtet getestet werden (vgl. Abschn. 7.3.6).

```
> aovRes <- aov(DV ~ IVbtw, data=mDf)
> cntrMat <- rbind("-0.5*(A+B)+C"=c(-1/2, -1/2, 1))      # Kontrastmatrix
> library(multcomp)                                         # für glht()
> summary(glht(aovRes, linfct=mcp(IVbtw=cntrMat), alternative="greater"))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = DV ~ IVbtw, data = mDf)
Linear Hypotheses:
Estimate Std. Error t value Pr(>t)
-0.5*(A+B)+C <= 0     0.2480     0.2369    1.047   0.152

# manuelle Kontrolle
> P   <- nlevels(mDf$IVbtw)                                # Anzahl Gruppen
> Nj  <- table(mDf$IVbtw)                                   # Gruppengrößen
> Mj  <- tapply(mDf$DV, mDf$IVbtw, mean)                 # Gruppenmittel
> SSw <- sum((mDf$DV - ave(mDf$DV, mDf$IVbtw, FUN=mean))^2) # QS within
> MSw <- SSw / (sum(Nj) - P)                               # mittlere QS within
```

```
> (psiHat <- sum(cntrMat[1, ] * Mj))                      # Kontrastschätzung
[1] 0.248

> lenSq  <- sum(cntrMat[1, ]^2 / Nj)                      # quadrierte Länge
> (tStat <- psiHat / sqrt(lenSq*MSw))                      # Teststatistik t
[1] 1.04672

> (pVal <- pt(tStat, sum(Nj) - P), lower.tail=FALSE)    # p-Wert
[1] 0.1522546
```

7.7.5 Erweiterung auf dreifaktorielles SPF- $p \cdot qr$ Design

Univariat formulierte Auswertung

Die varianzanalytische Auswertung eines dreifaktoriellen Designs mit einer Zwischen-Gruppen UV und zwei Intra-Gruppen Faktoren (SPF- $p \cdot qr$) unterscheidet sich nur wenig von jener im SPF- $p \cdot q$ Design. Abgesehen von der etwas komplizierteren Datenstruktur ist bei Verwendung von `aov()` nur die Modellformel mit den zu berücksichtigenden Effekten anzupassen.

```
> Nj <- 10                                         # Gruppengröße
> P  <- 2                                         # Stufen Zwischen-UV
> Q  <- 3                                         # Zeitpunkte Intra-UV1
> R  <- 2                                         # Zeitpunkte Intra-UV2
> id <- factor(rep(1:(P*Nj), times=Q*R))        # Blockzugehörigkeit
> IVbtw <- factor(rep(LETTERS[1:P], times=Q*R*Nj)) # Zwisch.-UV (between)
> IVwth1 <- factor(rep(1:Q, each=P*R*Nj))        # Intra-UV1 (within)
> IVwth2 <- factor(rep(rep(1:R, each=P*Nj), times=Q)) # Intra-UV2 (within)

# Simulation ohne Effekt IVbtw, mit Effekt IVwth1, IVwth2, ohne Interaktion
> DV_t11 <- round(rnorm(P*Nj, 8, 2), 2)          # AV zu t1-1
> DV_t21 <- round(rnorm(P*Nj, 13, 2), 2)          # AV zu t2-1
> DV_t31 <- round(rnorm(P*Nj, 13, 2), 2)          # AV zu t3-1
> DV_t12 <- round(rnorm(P*Nj, 10, 2), 2)          # AV zu t1-2
> DV_t22 <- round(rnorm(P*Nj, 15, 2), 2)          # AV zu t2-2
> DV_t32 <- round(rnorm(P*Nj, 15, 2), 2)          # AV zu t3-2
> DV     <- c(DV_t11, DV_t12, DV_t21, DV_t22, DV_t31, DV_t32)

# Datensatz im Long-Format und Auswertung mit aov() ...
> dfSPFp.qrl <- data.frame(id, IVbtw, IVwth1, IVwth2, DV)
> summary(aov(DV ~ IVbtw*IVwth1*IVwth2 + Error(id/(IVwth1*IVwth2)),
+             data=dfSPFp.qrl))
```

Als Maß für die Stärke jedes Effekts dient wie im SPF- $p \cdot q$ Design das generalisierte η_g^2 (vgl. Abschn. 7.7), das hier analog mit `EtaSq()` aus dem Paket `DescTools` geschätzt wird.

Multivariat formulierte Auswertung

Wird `Anova()` aus dem `car` Paket eingesetzt, ist zunächst zu Daten im Wide-Format überzugehen. Zudem sind Zwischen-Gruppen Design und Intra-Gruppen Struktur zu benennen.

```
# Datensatz im Wide-Format
> IVbtwW <- factor(rep(LETTERS[1:P], Nj)) # Zwischen-UV
> dfSPFp.qrW <- data.frame(IVbtwW, DV_t11, DV_t21, DV_t31, DV_t12,
+                             DV_t22, DV_t32)

# Zwischen-Gruppen Design
> fitSPFp.qr <- lm(cbind(DV_t11, DV_t21, DV_t31, DV_t12, DV_t22,
+                           DV_t32) ~ IVbtwW, data=dfSPFp.qrW)

# Intra-Gruppen Design
> inSPFp.qr <- expand.grid(IVwth1=gl(Q, 1), IVwth2=gl(R, 1))
> library(car) # für Anova()
> AnovaSPFp.qr <- Anova(fitSPFp.qr, idata=inSPFp.qr,
+                         idesign=~IVwth1*IVwth2)

> summary(AnovaSPFp.qr, multivariate=FALSE, univariate=TRUE) # ...
```

`Anova()` setzt hier voraus, dass die Anzahl der Freiheitsgrade des Blockeffekts ($(n_j - 1) \cdot p$ bei gleichen Gruppengrößen n_j) mindestens so groß wie die der Interaktion beider Intra-Gruppen Effekte ($(q - 1) \cdot (r - 1)$) ist. Es müssen also mindestens $\frac{(q-1)(r-1)}{p} + 1$ viele Blöcke pro Gruppe vorhanden sein. Ist dies nicht der Fall, kann für die automatische Berechnung der $\hat{\epsilon}$ -Korrekturen auf `anova()` ausgewichen werden (vgl. Abschn. 7.4.4, 7.7.3).

```
# Test Zwischen-Gruppen Faktor IVbtw ...
> anova(fitSPFp.qr, M=~1, X=~0, idata=inSPFp.qr, test="Spherical")

# Test Intra-Gruppen Faktor IVwth1, Interaktion IVbtw:IVwth1 ...
> anova(fitSPFp.qr, M=~IVwth1, X=~1, idata=inSPFp.qr, test="Spherical")

# Test Intra-Gruppen Faktor IVwth2, Interaktion IVbtw:IVwth2 ...
> anova(fitSPFp.qr, M=~IVwth1 + IVwth2, X=~IVwth1,
+         idata=inSPFp.qr, test="Spherical")

# Test Interaktionen IVwth1:IVwth2, IVbtw:IVwth1:IVwth2 ...
> anova(fitSPFp.qr, M=~IVwth1 + IVwth2 + IVwth1:IVwth2,
+         X=~IVwth1 + IVwth2, idata=inSPFp.qr, test="Spherical")

# R=2 -> Mauchly-Test IVwth2 hier unnötig
> mauchly.test(fitSPFp.qr, M=~IVwth1, X=~1, idata=inSPFp.qr)
> mauchly.test(fitSPFp.qr, M=~IVwth1 + IVwth2 + IVwth1:IVwth2,
+               X=~IVwth1 + IVwth2, idata=inSPFp.qr)
```

7.7.6 Erweiterung auf dreifaktorielles SPF- $pq \cdot r$ Design

Univariat formulierte Auswertung

Die varianzanalytische Auswertung eines dreifaktoriellen Designs mit zwei Zwischen-Gruppen UVn und einem Intra-Gruppen Faktor (SPF- $pq \cdot r$) mit `aov()` bringt ebenfalls eine etwas komplexere Datenstruktur mit sich und macht eine Anpassung der Modellformel notwendig.

```
> Njk      <- 10                                # Gruppengröße
> P        <- 2                                 # Stufen Zwischen-UV1
> Q        <- 2                                 # Stufen Zwischen-UV2
> R        <- 3                                 # Zeitpunkte Intra-UV
> id       <- factor(rep(1:(P*Q*Njk), times=R)) # Blockzugehörigkeit
> IVbtw1 <- factor(rep(1:P, times=Q*R*Njk))    # Zwischen-UV1 (between)
> IVbtw2 <- factor(rep(rep(1:Q, each=P*Njk), times=R)) # Zwischen-UV2
> IVwth   <- factor(rep(1:R, each=P*Q*Njk))     # Intra-UV (within)

# Simulation ohne Effekt IVbtw1, IVbtw2, mit Effekt IVwth, ohne Interaktion
> DV_t1 <- round(rnorm(P*Q*Njk, -3, 2), 2)      # AV zu t1
> DV_t2 <- round(rnorm(P*Q*Njk, 1, 2), 2)        # AV zu t2
> DV_t3 <- round(rnorm(P*Q*Njk, 2, 2), 2)        # AV zu t3
> DV     <- c(DV_t1, DV_t2, DV_t3)                # Gesamt-Daten

# Datensatz im Long-Format und Auswertung mit aov() ...
> dfSPFpq.rL <- data.frame(id, IVbtw1, IVbtw2, IVwth, DV)
> summary(aov(DV ~ IVbtw1*IVbtw2*IVwth + Error(id/IVwth), dfSPFpq.rL))
```

Als Maß für die Stärke jedes Effekts dient wie im SPF- $p \cdot q$ Design das generalisierte η_g^2 (vgl. Abschn. 7.7), das hier analog mit `EtaSq()` aus dem Paket `DescTools` geschätzt wird.

Multivariat formulierte Auswertung

Wird `Anova()` aus dem `car` Paket eingesetzt, ist zunächst zu Daten im Wide-Format überzugehen. Zudem sind Zwischen-Gruppen Design und Intra-Gruppen Struktur zu benennen.

```
# Datensatz im Wide-Format
> IVbtw1W    <- factor(rep(LETTERS[1:P], times=Q*Njk))
> IVbtw2W    <- factor(rep(c("+", "-"), each=P*Njk))
> dfSPFpq.rW <- data.frame(IV1W, IV2W, DV_t1, DV_t2, DV_t3)

# Zwischen-Gruppen Design
> fitSPFpq.r <- lm(cbind(DV_t1,DV_t2,DV_t3) ~ IVbtw1W*IVbtw2W, dfSPFpq.rW)
> inSPFpq.r  <- data.frame(IVwth=gl(R, 1))          # Intra-Gruppen Design
> library(car)                                       # für Anova()
> AnovaSPFpq.r <- Anova(fitSPFpq.r, idata=inSPFpq.r, idesign=~IVwth)
> summary(AnovaSPFpq.r, multivariate=FALSE, univariate=TRUE) # ...
```

`Anova()` setzt hier voraus, dass die Anzahl der Freiheitsgrade des Blockeffekts $((n_j - 1) \cdot p \cdot q)$ bei gleichen Gruppengrößen n_j mindestens so groß wie die des Intra-Gruppen Effekts $(r - 1)$ ist. Es müssen also mindestens $\frac{r-1}{p \cdot q} + 1$ viele Blöcke pro Gruppe vorhanden sein. Ist dies nicht der Fall, kann für die automatische Berechnung der ε -Korrekturen auf `anova()` ausgewichen werden (vgl. Abschn. 7.4.4, 7.7.3).

```
# Test IVbtw1, IVbtw2, IVbtw1:IVbtw2 ...
> anova(fitSPFpq.r, M=~1, X=~0, idata=inSPFpq.r, test="Spherical")

# Test IVwth, IVbtw1:IVwth, IVbtw2:IVwth, IVbtw1:IVbtw2:IVwth
> anova(fitSPFpq.r, M=~IVwth, X=~1, idata=inSPFpq.r, test="Spherical")
> mauchly.test(fitSPFpq.r, M=~IVwth, X=~1, idata=inSPFpq.r) # ...
```

7.8 Kovarianzanalyse

Bei der Kovarianzanalyse (ANCOVA) werden die Daten einer AV in einem linearen Modell aus quantitativen und kategorialen Variablen vorhergesagt. Sie stellt damit eine Kombination von linearer Regression und Varianzanalyse dar. Die Modellierung erfolgt mit den aus Regressions- und Varianzanalyse bekannten Funktionen, wobei in der $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$ Modellformel in der Rolle von $\langle \text{UV} \rangle$ sowohl quantitative Variablen wie Faktoren als Vorhersageterme auftauchen.

7.8.1 Test der Effekte von Gruppenzugehörigkeit und Kovariate

Im folgenden sei die Situation mit einer quantitativen und einer kategorialen UV vorausgesetzt. Die kategoriale UV wird dabei als Treatment-Variable, die quantitative Variable als Kovariate bezeichnet. Aus der Perspektive der Regression kann die Kovarianzanalyse als Prüfung betrachtet werden, ob die theoretischen Parameter des in jeder Treatment-Gruppe gültigen Regressionsmodells mit der Kovariate als Prädiktor und der AV als Kriterium identisch sind.

Aus der Perspektive der Varianzanalyse ist man an der Wirkung der Treatment-Variable interessiert, hält jedoch den Einfluss einer quantitativen Störvariable für möglich, der in jeder Treatment-Gruppe als linear angenommen wird. Da der Einfluss der Störvariable die Heterogenität der AV-Werte innerhalb jeder Gruppe erhöht, scheint es erstrebenswert, die Einflüsse der Treatment-UV und der Kovariate voneinander trennen zu können und so die power des Tests auf Gruppenunterschiede zu erhöhen. Meist wird die Fragestellung dahingehend eingeschränkt, dass in den Treatment-Gruppen nur unterschiedliche y -Achsenabschnitte der Regressionsgleichung zugelassen, die Steigungen dagegen als identisch vorausgesetzt werden. Die Treatment-Wirkung wäre dann in den Unterschieden der y -Achsenabschnitte erkennbar.

Als Beispiel diene jenes aus Maxwell und Delaney (2004, p. 429 ff.): An Depressionskranken sei ein Maß der Schwere ihrer Krankheit vor und nach einer therapeutischen Intervention erhoben worden, bei der es sich entweder um ein Medikament mit SSRI-Wirkstoff, um ein Placebo oder um den Verbleib auf einer Warteliste handelt. Die Vorher-Messung soll als Kovariate für die entscheidende Nachher-Messung dienen.

```
# Schweregrad in den Bedingungen bei Vorher- und Nachher-Messung
> SSRIpre <- c(18, 16, 16, 15, 14, 20, 14, 21, 25, 11) # SSRI vor
> SSRIpost <- c(12, 0, 10, 9, 0, 11, 2, 4, 15, 10) # SSRI nach
> PlacPre <- c(18, 16, 15, 14, 20, 25, 11, 25, 11, 22) # Placebo vor
> PlacPost <- c(11, 4, 19, 15, 3, 14, 10, 16, 10, 20) # Placebo nach
> WLpre <- c(15, 19, 10, 29, 24, 15, 9, 18, 22, 13) # Warteliste vor
> WLpost <- c(17, 25, 10, 22, 23, 10, 2, 10, 14, 7) # Warteliste nach
> P <- 3 # Anzahl Gruppen
> Nj <- rep(length(SSRIpre), times=P) # Gruppengrößen

# Faktor der Gruppenzugehörigkeiten
> IV <- factor(rep(1:3, Nj), labels=c("SSRI", "Placebo", "WL"))
> DVpre <- c(SSRIpre, PlacPre, WLpre) # alle prä-Messungen
> DVpost <- c(SSRIpost, PlacPost, WLpost) # alle post-Messungen
> dfAncova <- data.frame(id=1:sum(Nj), IV, DVpre, DVpost) # Datensatz
```

Als Veranschaulichung wird die Verteilung der Vorher- und Nachher-Werte in den Gruppen durch boxplots dargestellt, um daraufhin mit `anova()` die Varianzanalyse zunächst ohne, dann die Kovarianzanalyse mit Kovariate zu berechnen (Abb. 7.4, vgl. Abschn. 14.6.3). Dabei sei vorausgesetzt, dass der Steigungsparameter in allen Gruppen identisch ist, das Modell berücksichtigt also keinen Interaktionsterm von Treatment-Variable und Kovariate. Während der Gruppeneffekt ohne Kovariate nicht signifikant getestet wird, fällt sowohl der Test des Gruppeneffekts als auch jener der Kovariate in der Kovarianzanalyse signifikant aus.

```
> plot(DVpre ~ IV, main="Prä-Scores pro Gruppe") # Boxplot Prä
> plot(DVpost ~ IV, main="Post-Scores pro Gruppe") # Boxplot Post
> fitFull <- lm(DVpost ~ IV + DVpre, data=dfAncova) # vollst. Modell
> fitGrp <- lm(DVpost ~ IV, data=dfAncova) # ohne Kovariate
> fitRegr <- lm(DVpost ~ DVpre, data=dfAncova) # ohne Gruppe
> anova(fitGrp) # Test ohne Kovariate
Analysis of Variance Table
Response: DVpost
Df Sum Sq Mean Sq F value Pr(>F)
IV 2 240.47 120.233 3.0348 0.06473 .
Residuals 27 1069.70 39.619

> anova(fitFull) # Test mit Kovariate, QS Typ I
Analysis of Variance Table
Response: DVpost
Df Sum Sq Mean Sq F value Pr(>F)
IV 2 240.47 120.23 4.1332 0.027629 *
DVpre 1 313.37 313.37 10.7723 0.002937 **
Residuals 26 756.33 29.09
```

Zur Berechnung von Quadratsummen vom Typ III kann zum einen auf `Anova()` aus dem `car` Paket zurückgegriffen werden.¹⁸ Zum anderen lassen sich diese Quadratsummen mit `anova()`

¹⁸Da keine Interaktion von Treatment-Variable und Kovariate berücksichtigt wird, sind die Quadratsummen

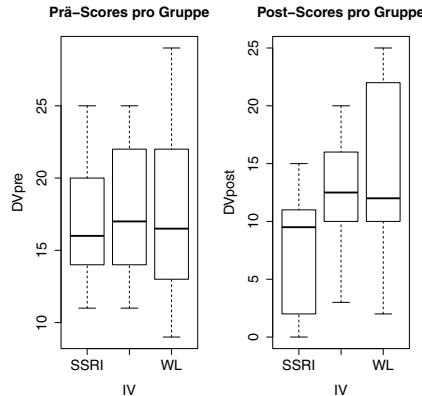


Abbildung 7.4: Kovarianzanalyse: Boxplots zum Vergleich der AV-Verteilungen in den Gruppen

durch den Test zweier geeigneter Modelle gegeneinander ermitteln (vgl. Abschn. 7.5.2): Für den Effekt der Kovariate sind dies auf der einen Seite das Modell ohne Kovariate als Vorhersageterm, auf der anderen Seite das vollständige Modell. Für den Effekt der Treatment-Variable entsprechend auf der einen Seite das Modell ohne Treatment-Variable als Vorhersageterm, auf der anderen Seite das vollständige Modell.

```
# Effektcodierung für Quadratsummen vom Typ III
> fitFiii <- lm(DVpost ~ IV + DVpre,
+                 contrasts=list(IV=contr.sum), data=dfAncova)

> library(car)                                     # für Anova()
> Anova(fitFiii, type="III")                      # QS Typ III
Anova Table (Type II tests)
Response: DVpost
          Sum Sq Df F value    Pr(>F)
(Intercept) 27.83  1  0.9567  0.337029
IV          217.15  2   3.7324  0.037584 *
DVpre       313.37  1 10.7723  0.002937 ** 
Residuals   756.33 26

# Quadratsummen vom Typ III über Modellvergleiche
> anova(fitRegr, fitFull)                         # Test Treatment ...
> anova(fitGrp,  fitFull)                         # Test Kovariate ...
```

Die Ergebnisse lassen sich auch manuell nachvollziehen, indem die Residual-Quadratsummen für das vollständige Modell, das Regressionsmodell ohne Treatment-Variable und das ANOVA-Modell ohne Kovariate berechnet werden. Die Effekt-Quadratsummen vom Typ III ergeben sich dann jeweils als Differenz der Residual-Quadratsumme des Modells, in dem dieser Effekt nicht berücksichtigt wird und der Residual-Quadratsumme des vollständigen Modells.

vom Typ II und III hier identisch.

```

> X    <- DVpre                      # kürzerer Name Kovariate
> Y    <- DVpost                     # kürzerer Name AV
> XMj <- tapply(X, IV, mean)         # Gruppenmittel Kovariate
> YMj <- tapply(Y, IV, mean)         # Gruppenmittel AV
> N    <- length(Y)                  # Gesamtanzahl Personen

# Residual-Quadratsumme vollständiges Modell
# zunächst gruppenweise zentrierte Daten der Kovariate und AV
> Xctr      <- X - ave(X, IV, FUN=mean)  # zentrierte Kovariate
> Yctr      <- Y - ave(Y, IV, FUN=mean)  # zentrierte AV
> bFull     <- cov(Xctr, Yctr) / var(Xctr) # b-Gewicht (alle Gruppen)
> aFull     <- YMj - bFull*XMj           # y-Achsenabschnitte
> YhatFull  <- bFull*X + aFull[IV]       # Vorhersage
> SSEfull   <- sum((Y-YhatFull)^2)        # Residual-QS
> dfSSEfull <- N-P-1                   # Freiheitsgrade Residual-QS
> MSEfull   <- SSEfull / dfSSEfull       # mittlere Residual-QS

# Residual-Quadratsumme Regressionsmodell ohne Treatment-Variablen
> bRegr     <- cov(X, Y) / var(X)        # b-Gewicht
> aRegr     <- mean(Y) - bRegr*mean(X)    # y-Achsenabschnitt
> YhatRegr  <- bRegr*X + aRegr          # Vorhersage
> SSEregr   <- sum((Y-YhatRegr)^2)        # Residual-QS
> dfSSEregr <- N-2                      # df Fehler
> MSEregr   <- SSEregr / dfSSEregr       # mittlere Residual-QS

# Residual-Quadratsumme ANOVA-Modell ohne Kovariate
> Vj      <- tapply(Y, IV, var)          # Gruppenvarianzen
> M       <- sum((Nj/N) * YMj)           # Gesamt-Mittel
> SSEgrp  <- sum((Nj-1) * Vj)            # Residual-QS
> dfSSEgrp <- N-P                      # df Fehler
> MSEgrp  <- SSEgrp / dfSSEgrp          # mittlere Residual-QS

# Effekt-Quadratsummen und F-Werte der zugehörigen Tests
> SSregr <- SSEgrp - SSEfull           # Effekt-QS Kovariate
> dfRegr <- dfSSEgrp - dfSSEfull       # df Kovariate -> 1
> MSregr <- SSregr / dfRegr             # MS Kovariate
> (Fregr <- MSregr / MSEfull)          # F-Wert Kovariate
[1] 10.77234

> SSgrp <- SSEregr - SSEfull           # Effekt-QS Treatment
> dfGrp <- dfSSEregr - dfSSEfull       # df Treatment -> P-1
> MSgrp <- SSgrp / dfGrp                # MS Treatment
> (Fgrp <- MSgrp / MSEfull)            # F-Wert Treatment
[1] 3.732399

```

Mit `summary(<lm-Modell>)` lassen sich die in den verschiedenen Gruppen angepassten Regressionsparameter ausgeben und einzeln auf Signifikanz testen.

```
> (sumRes <- summary(fitFull))
Call:
lm(formula = DVpost ~ IV + DVpre, data = dfAncova)

Residuals:
    Min      1Q  Median      3Q     Max 
-10.6842 -3.9615  0.6448  3.8773  9.9675 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) -3.6704    3.7525  -0.978  0.33703    
IVPlacebo     4.4483    2.4160   1.841  0.07703 .  
IVWL          6.4419    2.4133   2.669  0.01292 *  
DVpre         0.6453    0.1966   3.282  0.00294 ** 
---
Residual standard error: 5.393 on 26 degrees of freedom
Multiple R-squared: 0.4227, Adjusted R-squared: 0.3561 
F-statistic: 6.346 on 3 and 26 DF, p-value: 0.002252
```

Die unter **Coefficients** aufgeführten Testergebnisse sind so zu interpretieren, dass die SSRI-Gruppe als Referenzgruppe verwendet wurde, da sie die erste Faktorstufe in **IV** darstellt (vgl. Abschn. 2.6.5, 12.9.2). Ihre Koeffizienten finden sich in der Zeile **(Intercept)**. Für diese Gruppe ist der unter **Estimate** genannte Wert der *y*-Achsenabschnitt der Regressionsgerade. Die **Estimate** Werte für die Gruppen **Placebo** und **WL** geben jeweils die Differenz des *y*-Achsenabschnitts in dieser Gruppe zur Referenzgruppe an. Der in der letzten Spalte genannte *p*-Wert gibt Auskunft auf die Frage, ob dieser Unterschied signifikant von 0 verschieden ist. Die für alle Gruppen identische Steigung ist als **Estimate** für die Kovariate **DVpre** abzulesen. Ob sie signifikant von 0 verschieden ist, ergibt sich aus dem in der letzten Spalte genannten *p*-Wert.¹⁹

Eine grafische Veranschaulichung des linearen Zusammenhangs zwischen Vorher- und Nachher-Messwert in den einzelnen Gruppen erfolgt in Abb. 7.5.

```
# Steigung und y-Achsenabschnitte der Regressionsgeraden extrahieren
> coeffs <- coef(sumRes)                                # alle Koeffizienten
> iCptSSRI <- coeffs[1, 1]                             # b0 SSRI (Referenzgruppe)
> iCptPlac <- coeffs[2, 1] + iCptSSRI                # b0 Placebo
> iCptWL <- coeffs[3, 1] + iCptSSRI                  # b0 Warteliste
> slopeAll <- coeffs[4, 1]                               # gemeinsame Steigung

# Darstellungsbereich für x- und y-Achse wählen und Punktwolke darstellen
> xLims <- c(0, max(dfAncova$DVpre))
> yLims <- c(min(iCptSSRI, iCptPlac, iCptWL), max(dfAncova$DVpost))
> plot(DVpost ~ DVpre, data=dfAncova, xlim=xLims, ylim=yLims,
```

¹⁹ Die absolute Höhe der Gruppe der *y*-Achsenabschnitte lässt sich aus den Daten nicht unabhängig schätzen, während ihre Abstände untereinander eindeutig bestimmt sind. Die in Maxwell und Delaney (2004, p. 430 ff.) gezeigte Lösung fixiert den *y*-Achsenabschnitt der **WL** Gruppe auf 0 und berichtet die übrigen als Differenz dazu.

```

+      pch=rep(c(3, 17, 19), Nj), col=rep(c("red", "green", "blue"), Nj),
+      main="Rohdaten und Regressionsgerade pro Gruppe")

> legend(x="topleft", legend=levels(IV), pch=c(3, 17, 19),
+         col=c("red", "green", "blue"))

# Regressionsgeraden einfügen
> abline(iCeptSSRI, slopeAll, col="red")
> abline(iCeptPlac, slopeAll, col="green")
> abline(iCeptWL, slopeAll, col="blue")

```

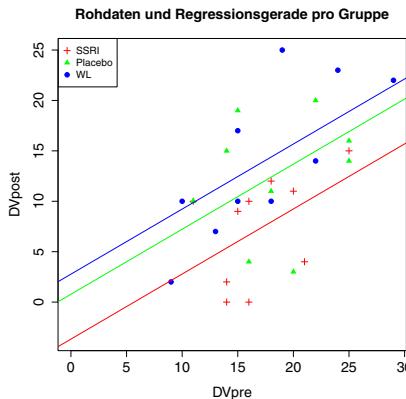


Abbildung 7.5: Kovarianzanalyse: nach Gruppen getrennte Regressionsgeraden

Als Maß für die Stärke jedes getesteten Effekts kann das partielle η_p^2 herangezogen werden, zu dessen Schätzung $\hat{\eta}_p^2$ jeweils seine Effekt-Quadratsumme an der Summe von ihr und der Residual-Quadratsumme relativiert wird. $\hat{\eta}_p^2$ der Kovariate ist hier gleich der quadrierten Partialkorrelation von Kovariate und Kriterium ohne die Treatment-Variable (vgl. Abschn. 6.7).

```

> SSEfull <- sum(residuals(fitFull)^2) # Fehler-Quadratsumme full model
> SSEgrp <- sum(residuals(fitGrp)^2) # Fehler-Quadratsumme Treatment
> SSEregr <- sum(residuals(fitRegr)^2) # Fehler-Quadratsumme Kovariate
> SSregr <- SSEgrp - SSEfull # Effekt-Quadratsumme Kovariate
> SSgrp <- SSEregr - SSEfull # Effekt-Quadratsumme Treatment
> SSregr / (SSregr + SSEfull) # partielle eta^2 Kovariate
[1] 0.2929469

# Kontrolle: Partialkorrelation Kovariate mit Kriterium ohne Faktor
> cor(residuals(lm(DVpre ~ IV)), residuals(lm(DVpost ~ IV)))^2
[1] 0.2929469

> SSgrp / (SSgrp + SSEfull) # partielle eta^2 Treatment
[1] 0.2230642

```

Sollen in der Kovarianzanalyse die Steigungen der Regressionsgeraden in den Gruppen nicht als identisch festgelegt, sondern auch bzgl. dieses Parameters Gruppenunterschiede i. S. einer Moderation (vgl. Abschn. 6.3.4) zugelassen werden, lautet das Modell:

```
> summary(lm(DVpost ~ IV + DVpre + IV:DVpre, data=dfAncova)) # ...
```

7.8.2 Beliebige a-priori Kontraste

Ähnlich wie bei Varianzanalysen lassen sich bei Kovarianzanalysen spezifische Vergleiche zwischen experimentellen Bedingungen in der Form von Kontrasten, also Linearkombinationen von Gruppenerwartungswerten testen (vgl. Abschn. 7.3.6). Die Kovariate findet dabei Berücksichtigung, indem hier der Vergleich zwischen korrigierten Erwartungswerten der AV stattfindet: Auf empirischer Ebene müssen für deren Schätzung zunächst die Regressionsparameter pro Gruppe bestimmt werden, wobei wie oben das *b*-Gewicht konstant sein und nur die Variation des *y*-Achsenabschnitts zwischen den Gruppen zugelassen werden soll. Der Gesamtmittelwert der Kovariate über alle Gruppen hinweg wird nun pro Gruppe in die ermittelte Regressionsgleichung eingesetzt. Das Ergebnis ist der korrigierte Gruppenmittelwert, der ausdrücken soll, welcher Wert in der AV zu erwarten wäre, wenn alle Personen denselben Wert auf der Kovariate (nämlich deren Gesamtmittelwert) hätten und sich nur in der Gruppenzugehörigkeit unterscheiden würden.

Im Beispiel werden drei Kontraste ohne α -Adjustierung getestet. Dafür ist es notwendig, die zugehörigen Kontrastvektoren als (ggf. benannte) Zeilen einer Matrix zusammenzustellen.

```
# Matrix der Kontrastkoeffizienten
> cntrMat <- rbind("SSRI-Placebo" = c(-1, 1, 0),
+                      "SSRI-WL"      = c(-1, 0, 1),
+                      "SSRI-0.5(P+WL)" = c(-2, 1, 1))

> library(multcomp) # für glht()
> aovAncova <- aov(DVpost ~ IV + DVpre, data=dfAncova)
> (sumRes <- summary(glht(aovAncova, linfct=mcp(IV=cntrMat),
+                           alternative="greater"), test=adjusted("none")))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = DVpost ~ IV + DVpre, data = dfAncova)
Linear Hypotheses:
Estimate Std. Error t value Pr(>|t|)
SSRI-Placebo <= 0     4.448     2.416   1.841  0.03852 *
SSRI-WL <= 0       6.442     2.413   2.669  0.00646 **
SSRI-0.5(P+WL) <= 0 10.890     4.183   2.603  0.00753 **
```

Die korrigierten Gruppenmittel lassen sich mit `effect("<UV>", (aov-Modell))` aus dem `effects` Paket (Fox, 2003) berechnen.

```
> library(effects) # für effect()
> YMjAdj <- effect("IV", aovAncova)
> summary(YMjAdj) # Konfidenzintervalle
```

```

IV effect
IV
    SSRI   Placebo      WL
7.536616 11.984895 13.978489

Lower 95 Percent Confidence Limits
IV
    SSRI   Placebo      WL
4.027629 8.476452 10.472608

Upper 95 Percent Confidence Limits
IV
    SSRI   Placebo      WL
11.04560 15.49334 17.48437

# manuelle Kontrolle
> YMjAdj     <- bFull*mean(X) + aFull          # korrig. Gruppenmittel
> psiHats    <- cntrMat    %*% YMjAdj        # Kontrastschätzungen
> lenSqs     <- cntrMat^2    %*% (1/Nj)       # quadrierte Längen
> Xctr       <- X - ave(X, IV, FUN=mean)      # zentrierte Kovariate
> fracs      <- (cntrMat %*% XMj)^2 / sum(Xctr^2)
> varPsiHats <- MSEfull * (lenSqs + fracs)    # Varianz der Schätzungen
> tStats     <- psiHats / sqrt(varPsiHats)    # Teststatistiken

# p-Werte einseitig
> pVals <- pt(abs(tStats), dfSSEfull, lower.tail=FALSE)
> data.frame(psiHats, tStats, pVals)
      psiHats      tStats      pVals
SSRI-Placebo  4.448279  1.8411994  0.038515271
SSRI-WL       6.441874  2.6692923  0.006461541
SSRI-0.5(P+WL) 10.890153  2.6031958  0.007529061

```

7.8.3 Beliebige post-hoc Kontraste nach Scheffé

Beliebige Kontraste können auch im Anschluss an eine signifikante Kovarianzanalyse getestet werden, die implizit simultan alle möglichen Kontraste prüft – spezifische Hypothesen liegen also bei ihrer Anwendung nicht vor. Aus diesem Grund muss im Anschluss an eine Kovarianzanalyse bei Einzeltests eine geeignete α -Adjustierung vorgenommen werden, hier vorgestellt nach der Methode von Scheffé.

Zunächst gilt für das Aufstellen eines Kontrasts alles bereits für beliebige a-priori Kontraste Ausgeführte. Lediglich die Wahl des kritischen Wertes weicht ab und ergibt sich zur α -Adjustierung aus einer *F*-Verteilung. Dieser kritische Wert ist mit dem Quadrat der a-priori *t*-Teststatistik zu vergleichen – die etwa in der von `summary(glht(...))` zurückgegebenen Liste in der Komponente `test$tstat` steht. Hier sollen dieselben Kontraste wie im a-priori Fall gerichtet getestet werden.

```
> dfSSgrp <- P-1                                # Freiheitsgrade Treatment-QS
> Fstats  <- sumRes$test$tstat^2                # quadrierte t-Teststatistiken

# p-Werte einseitig
> (pVals <- pf(Fstats/dfSSgrp, dfSSgrp, dfSSEfull, lower.tail=FALSE))
SSRI-Placebo      SSRI-WL   SSRI-0.5(P+WL)
0.20326188     0.04291472     0.04923999
```

7.9 Power, Effektstärke und notwendige Stichprobengröße

Mit Hilfe der Funktionen von Zufallsvariablen (vgl. Abschn. 5.3) lässt sich die power der vorgestellten inferenzstatistischen Tests berechnen, sofern eine exakte H_1 vorliegt. Hierfür ist es notwendig, zunächst auf Basis der Verteilung der Teststatistik unter H_0 mit der Quantilfunktion `q(Funktionsfamilie)()` den kritischen Wert für das gewünschte α -Niveau zu bestimmen. Mit Hilfe der zugehörigen Verteilungsfunktion `p(Funktionsfamilie)()` kann dann unter Gültigkeit der H_1 die power berechnet werden.

Analog lässt sich auch die notwendige Stichprobengröße (Fallzahl) ermitteln, für die der Test bei einem als gegeben vorausgesetzten Effekt eine gewisse power erreicht. Hierfür bedarf es meist eines Nonzentralitätsparameters der Verteilung der Teststatistik unter H_1 , der anhand der aus der Statistik bekannten Formeln zu berechnen ist und sich aus der theoretischen Effektstärke ergibt.

Die im Basisumfang von R enthaltenen Funktionen zur Bestimmung von power und Stichprobengröße besitzen eine recht eingeschränkte Funktionalität, so berücksichtigen sie nur Binomialtest, *t*-Test und Varianzanalyse. Mehr Möglichkeiten bieten die Pakete `pwr` (Champely, 2012) und `MBESS`. Insbesondere besitzt jedoch das kostenlose Programm G*Power (Faul, Erdfelder, Lang & Buchner, 2007) einen deutlich breiteren Einsatzbereich hinsichtlich der unterstützten Tests. Zudem bietet es vielfältige Möglichkeiten zur Visualisierung der Zusammenhänge von Effektstärke, power und Fallzahl.

7.9.1 Binomialtest

Im Beispiel soll zunächst der Fall eines rechtsseitigen Binomialtests betrachtet werden (vgl. Abschn. 10.1.1). Die Punktwahrscheinlichkeiten der einzelnen Ereignisse bei Gültigkeit von H_0 und H_1 sind zusammen mit dem kritischen Wert in Abb. 7.6 dargestellt.

```
> N      <- 7                                # Stichprobengröße
> pH0    <- 0.25                            # Wahrscheinlichkeit Treffer unter H0
> pH1    <- 0.7     # Wahrscheinlichkeit Treffer unter H1
> alpha  <- 0.05                            # Signifikanzniveau
> (critB <- qbinom(alpha, N, pH0, lower.tail=FALSE)) # krit. Wert
[1] 4

# power für konkrete H0, H1, N
> (powB <- pbinom(critB, N, pH1, lower.tail=FALSE))
```

```
[1] 0.6470695

> sum(dbinom((critB+1):N, N, pH1))      # Kontrolle: Summe Einzelwkt.
[1] 0.6470695

# Säulendiagramm: Veranschaulichung der Wahrscheinlichkeitsfunktionen
> dH0 <- dbinom(0:N, N, pH0)          # Verteilung unter H0
> dH1 <- dbinom(0:N, N, pH1)          # Verteilung unter H1
> mat <- rbind(dH0, dH1)
> rownames(mat) <- c("H0 (p=0.25)", "H1 (p=0.7)")
> colnames(mat) <- 0:N
> barsX <- barplot(mat, beside=TRUE, ylim=c(0, 0.35),
+                   xlab="Anzahl Treffer", ylab="Wahrscheinlichkeit",
+                   main="Binomialverteilung unter H0 und H1 (N=7)",
+                   col=c(rgb(1, 0.2, 0.2, 0.7), rgb(0.3, 0.3, 1, 0.6))),
+                   names.arg=colnames(mat), legend.text=rownames(mat))

> barplot(mat[, 1:(critB+1)], beside=TRUE, ylim=c(0, 0.35),
+           col=c("red", "blue"), add=TRUE)

# Hilfslinie für kritischen Wert
> xx <- barsX[2, critB+1] + (barsX[1, critB+2] - barsX[2, critB+1])/4
> abline(v=xx, col="green", lwd=2)
> text(xx-0.4, 0.34, adj=1, labels="kritischer Wert")
```

Beim Test mit der diskreten Binomialverteilung ist die power keine monotone Funktion der Stichprobengröße, sondern kann auch sinken, wenn der Test bei fester H_0 und H_1 mit Daten von mehr Beobachtungsobjekten durchgeführt wird. Dies liegt an der Veränderung des kritischen Wertes, die zusammen mit der Powerfunktion in Abb. 7.6 abgebildet ist.

```
> Nvec <- 2:15                      # betrachteter Bereich für N

# kritische Werte
> critBvec <- qbinom(alpha, size=Nvec, prob=pH0, lower.tail=FALSE)

# power
> powBvec <- pbinom(critBvec, size=Nvec, prob=pH1, lower.tail=FALSE)

# veranschauliche Verlauf der kritischen Werte
> par(mar=c(5, 4, 4, 4))            # breiterer rechter Rand
> plot(Nvec, critBvec, xlab="N", xaxt="n", yaxt="n", lwd=2, type="s",
+       pch=16, col="red", main="Powerfunktion und kritischer Wert",
+       Binomialtest", ylab="kritischer Wert")

# zeichne Achsen separat ein
> axis(side=1, at=seq(Nvec[1], Nvec[length(Nvec)], by=1))
> axis(side=2, at=seq(min(critBvec), max(critBvec), by=1), col="red")
```

```
> par(new=TRUE) # füge Powerfunktion hinzu
> plot(Nvec, powBvec, ylim=c(0, 1), type="b", lwd=2, pch=16,
+       col="blue", xlab=NA, ylab=NA, axes=FALSE)

# zeichne rechte Achse mit Achsenbeschriftung separat ein
> axis(side=4, at=seq(0, 1, by=0.1), col="blue")
> mtext(text="Power", side=4, line=3, cex=1.4)
```

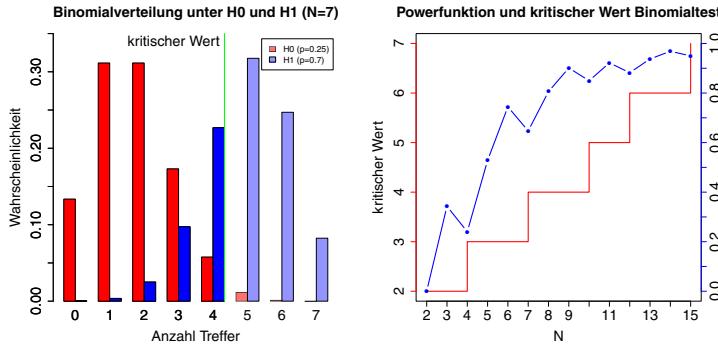


Abbildung 7.6: Binomialverteilung: Wahrscheinlichkeiten von Treffern unter H_0 und H_1 sowie kritischer Wert. Powerfunktion und kritischer Wert in Abhängigkeit von der Stichprobengröße

7.9.2 *t*-Test

Für den *t*-Test mit einer Stichprobe ist zur Bestimmung der Verteilung der Teststatistik unter H_1 die Berechnung des Nonzentralitätsparameters δ erforderlich, für den die theoretische Streuung sowie der Erwartungswert unter H_0 und H_1 bekannt sein muss.²⁰ Abbildung 7.7 zeigt die Verteilungen von t für das gegebene Hypothesenpaar und kennzeichnet die Flächen, deren Größe α , β und power bei einem rechtsseitigen Test entsprechen.

```
> N      <- 10          # Stichprobengröße
> muH0  <- 0           # Erwartungswert unter  $H_0$ 
> muH1  <- 1.6          # Erwartungswert unter  $H_1$ 
> alpha <- 0.05         # Signifikanzniveau
> sigma <- 2            # theoretische Streuung
> (d    <- (muH1-muH0) / sigma)   # Effektstärke d
[1] 0.8

> (delta <- (muH1-muH0) / (sigma/sqrt(N)))    # NZP, oder: sqrt(N)*d
[1] 2.529822
```

²⁰Für zwei unabhängige Stichproben mit Gruppengrößen n_1 und n_2 , Streuung σ und Erwartungswerten μ_1 und μ_2 unter H_1 ist $\delta = \sqrt{\frac{n_1 n_2}{n_1 + n_2}} \frac{\mu_2 - \mu_1}{\sigma}$. Für zwei abhängige Stichproben des jeweiligen Umfangs n mit theoretischen Streuungen σ_1 und σ_2 sowie der theoretischen Korrelation ρ ist $\delta = \sqrt{n} \frac{\mu_2 - \mu_1}{\sqrt{\sigma_1^2 + \sigma_2^2 + 2\rho\sigma_1\sigma_2}}$.

```

> (tCrit <- qt(1-alpha, N-1, lower.tail=FALSE)) # kritischer t-Wert
[1] 1.833113

> (powT <- pt(tCrit, N-1, delta, lower.tail=FALSE)) # power
[1] 0.7544248

# bestimme Werte der t-Verteilungen
> xLims <- c(-5, 10)
> tLeft <- seq(xLims[1], tCrit, length.out=100)
> tRight <- seq(tCrit, xLims[2], length.out=100)
> yH0r <- dt(tRight, N-1, 0)
> yH1l <- dt(tLeft, N-1, delta)
> yH1r <- dt(tRight, N-1, delta)

# markiere Flächen für alpha, beta und power
> curve(dt(x, N-1, 0), xlim=xLims, lwd=2, col="red", xlab="t", ylab="Dichte",
+        main="Verteilung von t unter H0 und H1", ylim=c(0, 0.4), xaxs="i")

> curve(dt(x, N-1, delta), lwd=2, col="blue", add=TRUE)
> polygon(c(tRight, rev(tRight)), c(yH0r, numeric(length(tRight))),
+           border=NA, col=rgb(1, 0.3, 0.3, 0.6))

> polygon(c(tLeft, rev(tLeft)), c(yH1l, numeric(length(tLeft))),
+           border=NA, col=rgb(0.3, 0.3, 1, 0.6))

> polygon(c(tRight, rev(tRight)), c(yH1r, numeric(length(tRight))),
+           border=NA, density=5, lty=2, lwd=2, angle=45, col="darkgray")

# zusätzliche Beschriftungen
> abline(v=tCrit, lty=1, lwd=3, col="red")
> text(tCrit+0.2, 0.4, adj=0, labels="kritischer Wert")
> text(tCrit-2.8, 0.3, adj=1, labels="Verteilung unter H0")
> text(tCrit+1.5, 0.3, adj=0, labels="Verteilung unter H1")
> text(tCrit+1.0, 0.08, adj=0, labels="Power")
> text(tCrit-0.7, 0.05, expression(beta))
> text(tCrit+0.5, 0.015, expression(alpha))

```

Wie für Binomial- und *t*-Tests demonstriert, kann die power analog für viele andere Tests manuell ermittelt werden. Für die Berechnung der power von *t*-Tests, bestimmten χ^2 -Tests und einfaktoriellen Varianzanalysen ohne Messwiederholung stehen in R auch eigene Funktionen bereit, deren Name nach dem Muster `power.<Test>.test()` aufgebaut ist. Diese Funktionen dienen gleichzeitig der Ermittlung der Stichprobengröße, die notwendig ist, damit ein Test bei fester Effektstärke eine vorgegebene Mindest-Power erzielt.²¹

```
> power.t.test(n, delta, sd, sig.level, power, strict=FALSE,
```

²¹Die Funktionen sind nicht vektorisiert, akzeptieren für jedes Argument also nur jeweils einen Wert.

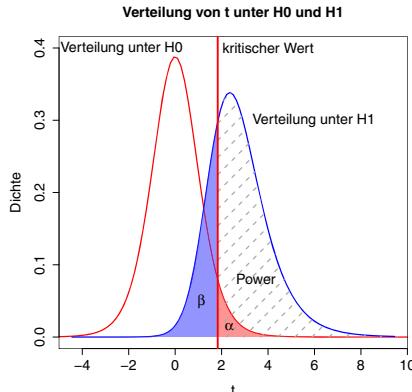


Abbildung 7.7: Rechtsseitiger *t*-Test für eine Stichprobe: Verteilung von *t* unter H_0 und H_1 , kritischer Wert, α , β und power

```
+ type=c("two.sample", "one.sample", "paired"),
+ alternative=c("two.sided", "one.sided"))
```

Von den Argumenten **n** für die Gruppengröße, **delta** für die Differenz der Erwartungswerte unter H_0 und H_1 , **sd** für die theoretische Streuung, **sig.level** für das α -Niveau und **power** für die power sind genau vier mit konkreten Werten zu nennen und eines auf **NULL** zu setzen. Das auf **NULL** gesetzte Argument wird dann auf Basis der übrigen berechnet. **n** bezieht sich im Fall zweier Stichproben auf die Größe jeder Gruppe – es werden also auch bei unabhängigen Stichproben gleiche Gruppengrößen vorausgesetzt.²²

Welche Art von *t*-Test vorliegt, kann über **type** angegeben werden, **alternative** legt fest, ob die H_1 gerichtet oder ungerichtet ist. Das Argument **strict** bestimmt, ob im zweiseitigen Test für die power die Wahrscheinlichkeit berücksichtigt werden soll, auch auf der falschen Seite (relativ zur Lage der tatsächlichen Verteilung unter H_1) die H_0 zu verwerfen.

Eine Fragestellung für den Einsatz von **power.t.test()** ist die Aufgabe, eine Stichprobengröße zu ermitteln, für die der Test bei einem als gegeben vorausgesetzten Effekt eine gewisse power erreicht. In diesem Fall ist also das Argument **n=NULL** zu übergeben, alle anderen sind zu spezifizieren. Die ausgegebene Gruppengröße ist i. d. R. nicht ganzzahlig, muss also in der konkreten Anwendung aufgerundet werden, wodurch sich die tatsächliche power des Tests leicht erhöht.

Im Beispiel soll für die oben gegebene Situation herausgefunden werden, wie viele Beobachtungsobjekte notwendig sind, damit der Test eine power von 0.9 besitzt.

²²Für unterschiedliche Gruppengrößen n_1 und n_2 lassen sich annähernd richtige Ergebnisse erzielen, wenn $2*((n1*n2)/(n1+n2))$ für das Argument **n** übergeben wird, wodurch die Berechnung des Nonzentralitätsparameters δ als $\text{sqrt}(n/2) * (\text{delta}/\text{sd})$ korrekt ist. Statt mit der richtigen Zahl der Freiheitsgrade $n_1 + n_2 - 2$ rechnet **power.t.test()** dann aber mit $4 \frac{n_1 n_2}{n_1 + n_2} - 2$. Der so entstehende Fehler wächst zwar mit der Differenz von n_1 und n_2 , bleibt jedoch absolut gesehen gering.

```
> power.t.test(n=NULL, delta=muH1-muH0, sd=sigma, sig.level=0.05,
+               power=0.9, type="one.sample", alternative="one.sided")
One-sample t test power calculation

  n = 14.84346
  delta = 1.6
  sd = 2
  sig.level = 0.05
  power = 0.9
alternative = one.sided
```

Eine andere Frage ist, wie groß bei einer gegebenen Stichprobengröße der tatsächliche Effekt sein muss, damit der Test eine bestimmte power erreicht. Hier ist `delta=NULL` zu übergeben, alle anderen Argumente sind zu spezifizieren.

7.9.3 Einfaktorielle Varianzanalyse

Die analog zu `power.t.test()` arbeitende Funktion für eine einfaktorielle Varianzanalyse ohne Messwiederholung lautet `power.anova.test()`.

```
> power.anova.test(groups, n, between.var, within.var, sig.level, power)
```

Von den Argumenten `groups` für die Anzahl der Gruppen, `n` für die Gruppengröße, `between.var` für die Varianz der Erwartungswerte unter H_1 ,²³ `within.var` für die theoretische Fehlervarianz, `sig.level` für den α -Fehler und `power` für die power sind genau fünf mit konkreten Werten zu nennen und eines auf `NULL` zu setzen. Das auf `NULL` gesetzte Argument wird dann auf Basis der übrigen berechnet. `n` bezieht sich auf die Größe jeder Gruppe – es werden also gleiche Gruppengrößen vorausgesetzt.

Im folgenden Beispiel einer einfaktoriellen Varianzanalyse ohne Messwiederholung soll die notwendige Stichprobengröße berechnet werden, damit der Test bei gegebenen Erwartungswerten unter H_1 eine bestimmte power erzielt.

```
> muJ <- c(100, 110, 115)                      # Erwartungswerte unter H1
> sigma <- 15                                     # theoretische Streuung
> power.anova.test(groups=3, n=NULL, between.var=var(muJ),
+                   within.var=sigma^2, sig.level=0.05, power=0.9)
Balanced one-way analysis of variance power calculation

  groups = 3
  n = 25.4322
  between.var = 58.33333
  within.var = 225
  sig.level = 0.05
  power = 0.9
```

²³Enthält der Vektor `muJ` die Erwartungswerte der Gruppen unter H_1 , muss wegen der in `power.anova.test()` verwendeten Formel für den Nonzentralitätsparameter λ die korrigierte Varianz der Erwartungswerte, also `var(muJ)`, für das Argument `between.var` übergeben werden.

NOTE: n is number in each group

Für das gegebene Beispiel folgt die manuelle Berechnung der power und der Maße für die Effektstärke einer Varianzanalyse im CR-p Design mit ungleichen Zellbesetzungen.

```
> P      <- length(muJ)                      # Anzahl Gruppen
> Nj    <- c(21, 17, 19)                     # Gruppengrößen
> N     <- sum(Nj)                          # Gesamt-N
> mu    <- sum(Nj * muJ) / N                # mittlerer EW, gewichtet
> alphaJ <- muJ - mu                        # Gruppeneffekte
> varMUj <- sum(Nj * alphaJ^2) / N          # Varianz der Erw.werte

# Maße für die Effektstärke - Bezeichnungen uneinheitlich
> (fSq <- varMUj / sigma^2)                  # f^2
[1] 0.1826887

> (etaSq <- varMUj / (sigma^2 + varMUj))   # eta^2 bzw. omega^2
[1] 0.1544690

# Nonzentralitätsparameter lambda bzw. delta^2 - Bezeichnungen uneinheitlich
> (lambda <- sum(Nj * alphaJ^2) / sigma^2)  # oder: N * fSq
[1] 10.41326

# kritischer Wert, alpha=0.05
> (Fcrit <- qf(0.05, P-1, N-P, lower.tail=FALSE))
[1] 3.168246

> (powF <- pf(Fcrit, P-1, N-P, lambda, lower.tail=FALSE))  # power
[1] 0.8090387

Lieg keine exakte  $H_1$  vor, ist die power als Funktion der Effektstärke  $f$  darstellbar (Abb. 7.8).

> fVals <- seq(0, 1.2, length.out=100)        # Effektstärken f auf x-Achse
> nn      <- seq(10, 25, by=5)                 # Stichprobengrößen: separate Kurven

# Funktion: berechne power für verschiedene Stichprobengrößen und f-Werte
> getFpow <- function(n) {
+   Fcrit <- qf(0.05, P-1, P*n - P, lower.tail=FALSE)
+   pf(Fcrit, P-1, P*n - P, P*n * fVals^2, lower.tail=FALSE)
+ }

> powsF <- sapply(nn, getFpow)      # Power-Werte für jede Stichprobengröße

# Beschriftungen vorbereiten
> yStr <- "Wahrscheinlichkeit der Annahme von  $H_1$ "
> mStr <- substitute(paste("Power als Funktion der Effektstärke f (",
```

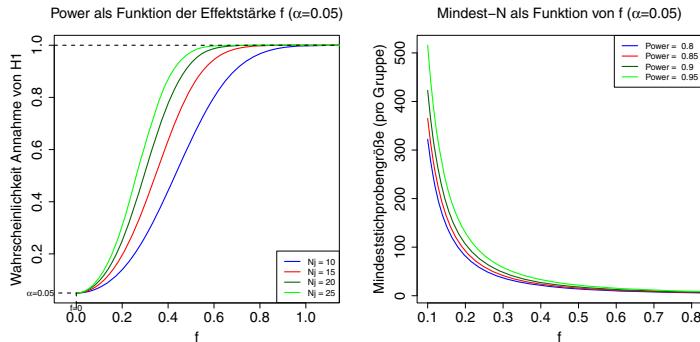


Abbildung 7.8: Einfaktorielle Varianzanalyse: power als Funktion der Effektstärke f für verschiedene Gruppengrößen sowie Mindeststichprobengröße als Funktion von f für verschiedene Power-Werte

```

+                               alpha, "=>0.05"))
# Power-Werte darstellen
> matplot(fVals, powsF, type="l", lty=1, lwd=2, xlab="f", ylab=yStr,
+           xlim=c(-0.05, 1.1), main=mStr,
+           col=c("blue", "red", "darkgreen", "green"))

# zusätzliche Beschriftungen
> lines(c(-2, 0, 0), c(0.05, 0.05, -2), lty=2, lwd=2)
> abline(h=1, lty=2, lwd=2)
> mtext("f=0", side=1, at=0)
> mtext(substitute(paste(alpha,"=>0.05",sep="")), side=2, at=0.05, las=1)
> legend(x="bottomright", legend=paste("Nj =", c(10, 15, 20, 25)), lwd=2,
+         col=c("blue", "red", "darkgreen", "green"))

```

Liegt keine exakte H_1 vor, lässt sich auch die Mindeststichprobengröße als Funktion der Effektstärke f darstellen (Abb. 7.8).

```

# Funktion, um für gegebene power und gegebene var.between
# die Mindeststichprobengröße zu berechnen
> getOneFn <- function(pp, varB) {
+   res <- power.anova.test(groups=P, n=NULL, between.var=varB,
+                           within.var=sigma^2, sig.level=0.05, power=pp)
+   res$n
+ }

# Funktion, um für mehrere Effektstärken f und mehrere Power-Werte
# gleichzeitig die Mindeststichprobengröße zu berechnen
# berechnet aus f zunächst var.between für power.anova.test()
> getManyFn <- function(ff, powF) {

```

```
+     varB <- ff^2 * (P/(P-1)) * sigma^2
+     sapply(powF, getOneFn, varB)
+ }

> fVals <- seq(0.1, 0.85, length.out=100) # Effektstärken f auf x-Achse
> pows <- seq(0.8, 0.95, by=0.05)        # Power-Werte: separate Kurven
> minN <- sapply(fVals, getManyFn, pows) # Mindeststichprobengrößen

# Beschriftungen vorbereiten
> yStr <- "Mindeststichprobengröße (pro Gruppe)"
> mStr <- substitute(paste("Mindest-N als Funktion von f (",
+                         alpha, "=0.05")))

# Mindeststichprobengrößen als Funktion von f darstellen
> matplot(fVals, t(minN), type="l", lty=1, lwd=2, xlab="f", ylab=yStr,
+           xlim=c(0.1, 0.8), main=mStr,
+           col=c("blue", "red", "darkgreen", "green"))

> legend(x="topright", legend=paste("Power = ", c(0.80, 0.85, 0.90,
+                                     0.95)), lwd=2, col=c("blue", "red", "darkgreen", "green"))
```

Kapitel 8

Regressionsmodelle für kategoriale Daten und Zähldaten

Das Modell der linearen Regression und Varianzanalyse (vgl. Abschn. 6.3, 7.3, 12.9.1) lässt sich zum verallgemeinerten linearen Modell (GLM, *generalized linear model*) erweitern, das auch für Daten einer kategorialen vorherzusagenden Variable Y geeignet ist.¹ Als Prädiktoren lassen sich sowohl kontinuierliche Variablen als auch Gruppierungsfaktoren einsetzen. Ein Spezialfall ist die logistische Regression für dichotome Y (codiert als 0 und 1). Im Vergleich zur Vorhersage quantitativer Variablen in der linearen Regression wird an diesem Beispiel zunächst folgende Schwierigkeit deutlich (für Details vgl. Agresti, 2007 sowie Fox & Weisberg, 2011):

Eine lineare Funktion von p Prädiktoren X_j der Form $\beta_0 + \beta_1 X_1 + \cdots + \beta_j X_j + \cdots + \beta_p X_p$ kann bei einem unbeschränkten Definitionsbereich beliebige Werte im Intervall $(-\infty, +\infty)$ annehmen. Es können sich durch die Modellgleichung also Werte ergeben, die weder von Y selbst, noch vom Erwartungswert $E(Y)$ angenommen werden können. Dabei ist $E(Y)$ für dichotome Y die Wahrscheinlichkeit eines Treffers $P(Y = 1)$, die im Intervall $[0, 1]$ liegt.

Dagegen ist die lineare Modellierung von $\text{logit}(P) = \ln \frac{P}{1-P}$ möglich, dem natürlichen Logarithmus des Wettquotienten, der Werte im Intervall $(-\infty, +\infty)$ annimmt. Setzt man $\ln \frac{P}{1-P} = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$ als gültiges Modell voraus und fasst die rechte Seite der Gleichung als *linearen Prädiktor $\mathbf{X}\boldsymbol{\beta}$* zusammen (vgl. Abschn. 12.9.1), so ist P mit der logistischen Funktion als Umkehrfunktion der Logit-Funktion als $P = \frac{e^{\mathbf{X}\boldsymbol{\beta}}}{1+e^{\mathbf{X}\boldsymbol{\beta}}} = \frac{1}{1+e^{-\mathbf{X}\boldsymbol{\beta}}}$ identifizierbar. Solche Transformationen des eigentlich vorherzusagenden Parameters, die eine lineare Modellierung ermöglichen, heißen *Link-Funktion $g(\cdot)$* . Sie müssen u. a. eine Umkehrfunktion $g^{-1}(\cdot)$ besitzen, so dass $g(E(Y)) = \mathbf{X}\boldsymbol{\beta}$ und $E(Y) = g^{-1}(\mathbf{X}\boldsymbol{\beta})$ gilt.

Verkürzt gesprochen wird im GLM anders als im allgemeinen linearen Modell nur $E(Y)$ über die Link-Funktion linear modelliert, nicht Y selbst. Es ist deshalb notwendig, die angenommene Form der bedingten Verteilung von Y für gegebene Prädiktorwerte explizit anzugeben (vgl. Abschn. 12.9). Mit dieser Form liegt auch die Varianz der Verteilung in Abhängigkeit von $E(Y)$ fest. Die bedingte Verteilung muss aus der natürlichen Exponentialfamilie stammen und wird mit einer Link-Funktion kombiniert, die $E(Y)$ mit dem linearen Prädiktor $\mathbf{X}\boldsymbol{\beta}$ in Beziehung setzt. Sind mehrere Link-Funktionen mit einer bedingten Verteilungsform kombinierbar, ist eine davon die *kanonische* Link-Funktion mit besonderen statistischen Eigenschaften.

Die lineare Regression ist ein Spezialfall des GLM, bei dem von bedingter Normalverteilung von Y ausgegangen wird (vgl. Abschn. 12.9.4, Abb. 12.5) und $E(Y)$ zudem nicht durch eine Link-Funktion transformiert werden muss, um linear modellierbar zu sein. Anders als in der

¹Abschnitt 6.6.4 gibt Hinweise auf gemischte Regressionsmodelle und verallgemeinerte Schätzgleichungen (GEE) für abhängige Daten – etwa durch Messwiederholung oder Clusterung, die analog auf kategoriale Zielgrößen übertragen werden können.

linearen Regression werden die Parameter β_j im GLM über die Maximum-Likelihood-Methode geschätzt (vgl. Abschn. 8.1.7). In den folgenden Abschnitten sollen die logistische, ordinale, multinomiale und Poisson-Regression ebenso vorgestellt werden wie log-lineare Modelle. Für die Visualisierung kategorialer Daten vgl. Abschn. 14.4 sowie das Paket `vcf` (Meyer, Zeileis & Hornik, 2014).

8.1 Logistische Regression

In der logistischen Regression für dichotome Daten wird als bedingte Verteilung von Y die Binomialverteilung angenommen, die kanonische Link-Funktion ist die Logit-Funktion. Die bedingten Verteilungen sind dann durch $E(Y) = P$ vollständig festgelegt, da ihre Varianz gleich $n_i P(1 - P)$ ist. Dabei ist n_i die Anzahl der dichotomen Messwerte für dieselbe Kombination von Prädiktorwerten.

8.1.1 Modell für dichotome Daten anpassen

Die Anpassung einer logistischen Regression geschieht mit der `glm()` Funktion, mit der allgemein GLM-Modelle spezifiziert werden können.² Hier sei zunächst der Fall betrachtet, dass die erhobenen Daten als dichotome Variable vorliegen.

```
> glm(formula=<Modellformel>, family=<Verteilungsfamilie>, data=<Datensatz>)
```

Unter `formula` wird eine Modellformel $\langle AV \rangle \sim \langle Prädiktoren \rangle$ wie mit `lm()` formuliert, wobei sowohl quantitative wie kategoriale Variablen als Prädiktoren möglich sind. Die AV muss ein Objekt der Klasse `factor` mit zwei Stufen sein, die Auftretenswahrscheinlichkeit P bezieht sich auf die zweite Faktorstufe. Weiter ist unter `family` ein Objekt anzugeben, das die für die AV angenommene bedingte Verteilung sowie die Link-Funktion benennt (vgl. `?family`). Im Fall der logistischen Regression ist dieses Argument auf `binomial(link="logit")` zu setzen.

Als Beispiel sei jenes aus der Kovarianzanalyse herangezogen (vgl. Abschn. 7.8), wobei hier vorhergesagt werden soll, ob die Depressivität nach der Behandlung über dem Median liegt. Die Beziehung zwischen der Depressivität vor und nach der Behandlung in den drei Gruppen soll dabei zunächst über ein Diagramm mit dem nonparametrisch geschätzten Verlauf der Trefferwahrscheinlichkeit veranschaulicht werden, das `cdplot(<Modellformel>)` erzeugt (*conditional density plot*, Abb. 8.1).

```
# Median-Split zur Umwandlung der quantitativen AV in dichotomen Faktor
> dfAncova <- transform(dfAncova, postFac=cut(DVpost,
+                                         breaks=c(-Inf, median(DVpost), Inf),
+                                         labels=c("lo", "hi")))

# conditional density plot der AV in den drei Gruppen
> cdplot(postFac ~ DVpre, data=dfAncova, subset=IV == "SSRI",
+         main="Geschätzte Kategorien-Wkt. SSRI")
```

²Für die bedingte logistische Regression bei Stratifizierung der Beobachtungen vgl. `clogit()` aus dem Paket `survival` (Therneau, 2014).

```
> cdplot(postFac ~ DVpre, data=dfAncova, subset=IV == "Placebo",
+         main="Geschätzte Kategorien-Wkt. Placebo")

> cdplot(postFac ~ DVpre, data=dfAncova, subset=IV == "WL",
+         main="Geschätzte Kategorien-Wkt. WL")
```

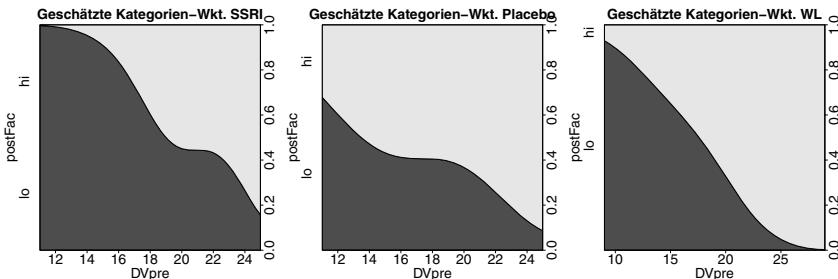


Abbildung 8.1: Nonparametrisch geschätzte Kategorienwahrscheinlichkeiten in Abhängigkeit vom Prädiktor in drei Behandlungsgruppen

```
# Anpassung des logistischen Regressionsmodells
> (glmFit <- glm(postFac ~ DVpre + IV, family=binomial(link="logit"),
+                  data=dfAncova))
Call: glm(formula = postFac ~ DVpre + IV,
          family = binomial(link = "logit"), data = dfAncova)
```

```
Coefficients:
(Intercept) DVpre IVPlacebo IVWL
-8.4230    0.4258    1.7306   1.2027
```

```
Degrees of Freedom: 29 Total (i.e. Null); 26 Residual
Null Deviance: 41.46
Residual Deviance: 24.41 AIC: 32.41
```

Die Ausgabe nennt unter der Überschrift **Coefficients** zunächst die Schätzungen b_j der Modellparameter β_j der logistischen Regression, wobei der in der Spalte **(Intercept)** aufgeführte Wert die Schätzung b_0 ist. Der Parameter eines Prädiktors ist als Ausmaß der Änderung der Vorhersage $\ln \frac{\hat{P}}{1-\hat{P}}$ zu interpretieren, wenn der Prädiktor X_j um eine Einheit wächst, also als Differenz der logits (vgl. Abschn. 7.8.1 zur Bedeutung der zwei mit dem Faktor IV assoziierten Parameter).

Einfacher ist die Bedeutung eines exponentzierten Parameters e^{b_j} zu erfassen: Dieser Koeffizient gibt an, um welchen Faktor der vorhergesagte Wettquotient $\frac{\hat{P}}{1-\hat{P}}$ zunimmt, wenn sich X_j um eine Einheit vergrößert.³ Dies ist das Verhältnis des vorhergesagten Wettquotienten nach der Änderung um eine Einheit zum Wettquotienten vor dieser Änderung, also ihr odds ratio (vgl.

³Für einen Prädiktor X : $\left(\frac{\hat{P}}{1-\hat{P}}\right)_{X+1} = e^{b_0+b_1(X+1)} = e^{b_0} e^{b_1(X+1)} = e^{b_0} e^{b_1 X} e^{b_1} = e^{b_1} e^{b_0+b_1 X} = e^{b_1} \left(\frac{\hat{P}}{1-\hat{P}}\right)_X$.

Abschn. 10.2.6). Dagegen besitzt e^{β_0} keine intuitive Bedeutung. Wie bei linearen Modellen extrahiert `coef(glm-Modell)` die Parameterschätzungen.

```
> exp(coef(glmFit))          # exponenzierte Koeffizienten
(Intercept)      DVpre       IVPPlacebo      IVWL
0.0002197532  1.5308001795  5.6440022784  3.3291484767
```

Wie bei der linearen Regression lassen sich die Konfidenzintervalle für die wahren Parameter mit `confint(glm-Modell)` berechnen.⁴ Für die Konfidenzintervalle der odds ratios e^{β_j} sind die Intervallgrenzen zu exponentiieren.

```
> exp(confint(glmFit))      # zugehörige Konfidenzintervalle
   2.5 %     97.5 %
(Intercept) 1.488482e-07  0.0251596
DVpre        1.193766e+00  2.2446549
IVPPlacebo  5.343091e-01  95.1942030
IVWL         2.916673e-01  52.2883653
```

8.1.2 Modell für binomiale Daten anpassen

Logistische Regressionen können mit `glm()` auch dann angepasst werden, wenn pro Kombination i von Prädiktorwerten mehrere dichotome Werte erhoben und bereits zu Ausprägungen einer binomialverteilten Variable aufsummiert wurden ($n_i \geq 1$). In diesem Fall ist das Kriterium in Form einer Matrix an die Modellformel zu übergeben: Jede Zeile der Matrix steht für eine Kombination von Prädiktorwerten i , für die n_i dichotome Werte vorhanden sind. Die erste Spalte der Matrix nennt die Anzahl der Treffer, die zweite Spalte die Anzahl der Nicht-Treffer. Beide Spalten summieren sich pro Zeile also zu n_i .

```
> N      <- 100          # Anzahl Kombinationen
> x1    <- rnorm(N, 100, 15)  # Prädiktor 1
> x2    <- rnorm(N, 10, 3)    # Prädiktor 2
> total <- sample(40:60, N, replace=TRUE) # ni
> hits  <- rbinom(N, total, prob=0.4)    # Treffer pro Kombination
> hitMat <- cbind(hits, total-hits)      # Matrix Treffer, Nicht-Treffer

# logistische Regression für absolute Häufigkeiten anpassen
> glm(hitMat ~ x1 + x2, family=binomial(link="logit"))      # ...
```

Liegt die AV als Vektor relativer Häufigkeiten eines Treffers vor, sind zusätzlich die n_i als Vektor an das Argument `weights` von `glm()` zu übergeben.

```
> relHits <- hits/total      # relative Häufigkeiten ...
> glm(relHits ~ x1 + x2, weights=total, family=binomial(link="logit"))
```

⁴Die so ermittelten Konfidenzintervalle basieren auf der Profile-Likelihood-Methode und sind asymmetrisch. Demgegenüber berechnet `confint.default(glm-Modell)` symmetrische Wald-Konfidenzintervalle, die asymptotische Normalverteilung der Parameterschätzungen voraussetzen.

8.1.3 Anpassungsgüte

Als Maß für die Güte der Modellpassung wird von `glm()` die Residual-Devianz D als Summe der quadrierten Devianz-Residuen ausgegeben.⁵ Mit \hat{L} als geschätzter likelihood des Modells, die an der likelihood eines Modells mit perfekter Vorhersage normalisiert wurde, gilt $D = -2 \ln \hat{L}$. Durch die Maximum-Likelihood-Schätzung der Parameter wird \hat{L} maximiert, D also minimiert – analog zur Fehlerquadratsumme in der linearen Regression (vgl. Abschn. 6.2).⁶ Weiter erhält man den Wert des Informationskriteriums $AIC = D + 2(p + 1)$, bei dem ebenfalls kleinere Werte für eine bessere Anpassung sprechen (vgl. Abschn. 6.3.3). Dabei ist $p + 1$ die Anzahl zu schätzender Parameter (p Gewichte β_j sowie β_0).⁷

Die Residual-Devianz eines Modells ermittelt `deviance(glm-Modell)`, die logarithmierte geschätzte likelihood eines Modells `logLik(glm-Modell)` und den AIC-Wert `AIC(glm-Modell)`.

```
> (Dev <- deviance(glmFit))                                # Devianz
[1] 24.40857

> sum(residuals(glmFit)^2)                                 # Devianz
[1] 24.40857

> as.vector(-2 * logLik(glmFit))                          # Devianz
[1] 24.40857

> all.equal(AIC(glmFit), Dev + 2*(3+1))                  # AIC
[1] TRUE
```

Weitere Maße der Anpassungsgüte sind pseudo- R^2 -Koeffizienten, die an den Determinationskoeffizienten in der linearen Regression angelehnt sind (vgl. Abschn. 6.2.2).⁸ Die Varianten nach McFadden, Cox & Snell und Nagelkerke können auf Basis der Ausgabe von `glm()` manuell ermittelt werden.⁹ Hier soll \hat{L}_0 die geschätzte likelihood des 0-Modells ohne Prädiktoren X_j mit nur dem Parameter β_0 sein. Analog sei \hat{L}_f die geschätzte likelihood des Modells mit allen berücksichtigten Prädiktoren. Ferner bezeichne D_0 bzw. D_f die jeweils zugehörige Devianz.

- $R_{\text{McFadden}}^2 = 1 - \frac{\ln \hat{L}_f}{\ln \hat{L}_0} = 1 - \frac{D_f}{D_0}$
 - $R_{\text{Cox \& Snell}}^2 = 1 - \left(\frac{\hat{L}_0}{\hat{L}_f} \right)^{\frac{2}{N}} = 1 - e^{(\ln \hat{L}_0 - \ln \hat{L}_f) \frac{2}{N}}$
- Das Maximum von $R_{\text{Cox \& Snell}}^2$ beträgt $1 - \hat{L}_0^{\frac{2}{N}} < 1$.

⁵In der Voreinstellung gibt `residuals(glm-Modell, type="Typ")` Devianz-Residuen aus. Für andere Residuen-Varianten kann das Argument `type` verwendet werden (vgl. `?residuals.glm`).

⁶Für die gewöhnliche lineare Regression stimmen Devianz und Fehlerquadratsumme überein.

⁷Bei der gewöhnlichen linearen Regression wie auch bei der logistischen Regression mit der quasi-binomial Familie (s. u.) ist zusätzlich ein Varianzparameter zu schätzen. Hier beträgt die Anzahl also $p + 1 + 1$.

⁸Anders als in der linearen Regression lassen sich die pseudo- R^2 -Maße jedoch nicht als Verhältnis von Variabilitäten verstehen. Ihre Vergleichbarkeit über verschiedene Datensätze hinweg ist zudem eingeschränkt – so beziehen etwa $R_{\text{Cox \& Snell}}^2$ sowie $R_{\text{Nagelkerke}}^2$ neben der absoluten Anpassung auch die Stichprobengröße ein.

⁹Für weitere Gütemaße der Modellanpassung vgl. die Funktion `lrm()` aus dem Paket `rms`, die neben Nagelkerkes pseudo- R^2 die Fläche unter der ROC-Kurve (vgl. Abschn. 10.2.7) ebenso bestimmt wie etwa Somers' d , Goodman und Kruskals γ sowie Kendalls τ für die vorhergesagten Wahrscheinlichkeiten und beobachteten Werte (vgl. Abschn. 10.3.1).

- $R^2_{\text{Nagelkerke}} = R^2_{\text{Cox \& Snell}} / (1 - \hat{L}_0^{2/N})$

Wie bei linearen Modellen lässt sich ein bereits angepasstes Modell mit `update(glm-Modell)` ändern (vgl. Abschn. 6.3.2), etwa alle Prädiktoren bis auf den absoluten Term entfernen.

```
> glm0 <- update(glmFit, . ~ 1)                      # O-Modell
> LL0  <- logLik(glm0)      # gesch. log-likelihood O-Modell
> LLf  <- logLik(glmFit)      # gesch. log-likelihood vollständiges Modell
> as.vector(1 - (LLf / LL0))                  # R^2 McFadden
[1] 0.411209

> N <- nobs(glmFit)          # Anzahl der Beobachtungen
> as.vector(1 - exp((2/N) * (LL0 - LLf)))      # R^2 Cox & Snell
[1] 0.4334714

# R^2 Nagelkerke
> as.vector((1 - exp((2/N) * (LL0 - LLf))) / (1 - exp(LL0^(2/N))))
[1] 0.578822
```

Eine Alternative zu pseudo- R^2 -Koeffizienten ist der Diskriminationsindex von Tjur. Er berechnet sich als Differenz der jeweils mittleren vorhergesagten Trefferwahrscheinlichkeit für die Beobachtungen mit Treffer und Nicht-Treffer. Die vorhergesagte Wahrscheinlichkeit \hat{P} erhält man etwa mit `fitted(glm-Modell)` (vgl. Abschn. 8.1.4). Der Diskriminationsindex nimmt maximal den Wert 1 an, ist jedoch anders als die pseudo- R^2 -Koeffizienten nicht auf andere Regressionsmodelle für diskrete Kriterien verallgemeinerbar.

```
> Phat <- fitted(glmFit)    # vorhergesagte Trefferwahrscheinlichkeiten

# mittlere vorhergesagte Wahrscheinlichkeit für wFac = lo und wFac = hi
> (PhatLoHi <- aggregate(Phat ~ postFac, FUN=mean, data=dfAncova))
  postFac      Phat
1      lo 0.2521042
2      hi 0.7118809

# Tjur Diskriminationsindex
> diff(PhatLoHi$Phat)      # Differenz mittlere vorherges. Wkt.
[1] 0.4597767
```

Für die Kreuzvalidierung verallgemeinerter linearer Modellen vgl. Abschn. 13.2. Methoden zur Diagnose von Ausreißern in den Prädiktoren können aus der linearen Regression ebenso übernommen werden wie Cooks Distanz und der Index DfBETAS zur Identifikation einflussreicher Beobachtungen (vgl. Abschn. 6.5.1). Anders als im Modell der linearen Regression hängt im GLM die Varianz vom Erwartungswert ab. Daher sollte eine Residuen-Diagnostik mit spread-level oder scale-location plots (vgl. Abschn. 6.5.2) Devianz-Residuen verwenden, die man mit `residuals(glm-Modell)` erhält. Für Methoden zur Einschätzung von Multikollinearität der Prädiktoren vgl. Abschn. 6.5.3.

8.1.4 Vorhersage, Klassifikation und Anwendung auf neue Daten

Die vorhergesagte Wahrscheinlichkeit $\hat{P} = \frac{1}{1+e^{-\mathbf{x}\beta}}$ berechnet sich für jede Beobachtung durch Einsetzen der Parameterschätzungen b_j in $\mathbf{X}\hat{\beta} = b_0 + b_1X_1 + \dots + b_pX_p$. Man erhält sie mit `fitted(<glm-Modell>)` oder mit der Funktion `predict(<glm-Modell>, type="response")`. In der Voreinstellung `type="link"` werden die vorhergesagten Logit-Werte ausgegeben.

```
> Phat <- fitted(glmFit)           # vorhergesagte Wahrscheinlichkeit
> predict(glmFit, type="response") # äquivalent ...
> logitHat <- predict(glmFit, type="link")        # vorhergesagte Logits
> all.equal(logitHat, log(Phat / (1-Phat)))       # Kontrolle: log-odds
[1] TRUE
```

Mit der gewählten Herangehensweise ist die mittlere vorhergesagte Wahrscheinlichkeit gleich der empirischen relativen Häufigkeit eines Treffers.¹⁰

```
> mean(Phat)                      # mittlere vorhergesagte W'keit
[1] 0.4666667

# relative Trefferhäufigkeit
> prop.table(xtabs(~ postFac, data=dfAncova))
postFac
      lo      hi 
0.5333333 0.4666667
```

Die vorhergesagte Trefferwahrscheinlichkeit soll hier als Grundlage für eine dichotome Klassifikation mit der Schwelle $\hat{P} = 0.5$ verwendet werden. Diese Klassifikation kann mit den tatsächlichen Kategorien in einer Konfusionsmatrix verglichen und etwa die Rate der korrekten Klassifikation berechnet werden.¹¹

```
# Klassifikation auf Basis der Vorhersage
> thresh <- 0.5                  # Schwelle bei P=0.5
> facHat <- cut(Phat, breaks=c(-Inf, thresh, Inf), labels=c("lo", "hi"))

# Kontingenztafel: tatsächliche vs. vorhergesagte Kategorie
> cTab <- xtabs(~ postFac + facHat, data=dfAncova)
> addmargins(cTab)
facHat
postFac lo hi Sum
      lo 12  4 16
      hi  4 10 14
      Sum 16 14 30
```

¹⁰Dies ist der Fall, wenn die kanonische Link-Funktion und Maximum-Likelihood-Schätzungen der Parameter gewählt werden und das Modell einen absoluten Term β_0 beinhaltet.

¹¹Vergleiche Abschn. 13.2 für die Kreuzvalidierung zur Abschätzung der Vorhersagegüte in neuen Stichproben sowie Abschn. 10.2.6, 10.2.7, 10.3.3 für weitere Möglichkeiten, Klassifikationen zu analysieren. Vergleiche Abschn. 12.8 für die Diskriminanzanalyse sowie die dortige Fußnote 44 für Hinweise zu weiteren Klassifikationsverfahren.

```
> (CCR <- sum(diag(cTab)) / sum(cTab)) # Rate der korrekten Klass.
[1] 0.7333333
```

Inwieweit die Vorhersage der logistischen Regression zutrifft, kann auf unterschiedliche Weise grafisch veranschaulicht werden. Eine Möglichkeit stellt die vorhergesagten logits $\ln \frac{\hat{P}}{1-\hat{P}}$ dar und hebt die tatsächlichen Treffer farblich hervor (Abb. 8.2). Vorhergesagte logits > 0 sind bei einer Schwelle von $\hat{P} = 0.5$ äquivalent zur Vorhersage eines Treffers, so dass die tatsächlichen Treffer hier bei einer guten Klassifikation oberhalb einer Referenzlinie bei 0 liegen sollten.

```
> plot(logitHat, pch=c(1, 16)[unclass(regDF$wFac)], cex=1.5, lwd=2,
+       main="(Fehl-) Klassifikation durch Vorhersage",
+       ylab="vorhergesagte logits")

> abline(h=0) # Referenzlinie
> legend(x="bottomright", legend=c("lo", "hi"), pch=c(1, 16), cex=1.5,
+         lty=NA, lwd=2, bg="white")
```

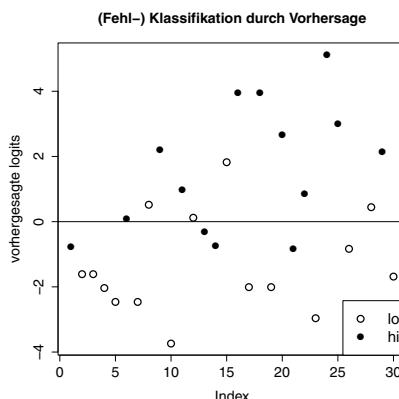


Abbildung 8.2: Vorhersage und Daten einer logistischen Regression. Daten, die auf Basis der Vorhersage falsch klassifiziert würden, sind farblich hervorgehoben

An das Argument `newdata` von `predict()` kann zusätzlich ein Datensatz übergeben werden, der neue Daten für Variablen mit denselben Namen, und bei Faktoren zusätzlich denselben Stufen wie jene der ursprünglichen Prädiktoren enthält. Als Ergebnis erhält man die vorhergesagten Trefferwahrscheinlichkeiten für die neuen Prädiktorwerte (vgl. Abschn. 6.4).

```
> Nnew <- 3 # Anzahl neuer Daten
> dfNew <- data.frame(DVpre=rnorm(Nnew, 20, sd=7),
+                         IV=factor(rep("SSRI", Nnew),
+                                     levels=levels(dfAncova$IV)))

> predict(glmFit, newdata=dfNew, type="response") # Vorhersage
1           2           3
0.947324911 0.008185519 0.123296946
```

8.1.5 Signifikanztests für Parameter und Modell

Die geschätzten Gewichte b lassen sich mit `summary(glm-Modell)` einzeln einem Wald-Signifikanztest unterziehen. In der Ausgabe finden sich dazu unter der Überschrift **Coefficients** die Gewichte b in der Spalte **Estimate**, deren geschätzte Streuungen $\hat{\sigma}_b$ in der Spalte **Std. Error** und die zugehörigen z -Werte $\frac{b}{\hat{\sigma}_b}$ in der Spalte **z value**. Der Test setzt voraus, dass z unter der Nullhypothese asymptotisch standardnormalverteilt ist. Die zugehörigen p -Werte stehen in der Spalte **Pr(>|z|)**.

Call:

```
glm(formula = postFac ~ DVpre + IV, family = binomial(link = "logit"),
     data = dfAncova)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-1.9865	-0.5629	-0.2372	0.4660	1.5455

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	-8.4230	2.9502	-2.855	0.0043 **
DVpre	0.4258	0.1553	2.742	0.0061 **
IVPlacebo	1.7306	1.2733	1.359	0.1741
IVWL	1.2027	1.2735	0.944	0.3450

(Dispersion parameter for binomial family taken to be 1)

```
Null deviance: 41.455 on 29 degrees of freedom
Residual deviance: 24.409 on 26 degrees of freedom
AIC: 32.409
```

Number of Fisher Scoring iterations: 5

Geeigneter als Wald-Tests sind oft Likelihood-Quotienten-Tests der Parameter, die auf der asymptotisch χ^2 -verteilten Devianz-Differenz zweier nested Modelle mit demselben Kriterium beruhen:¹² Der Prädiktorensatz des eingeschränkten Modells ist dabei vollständig im Prädiktorensatz des umfassenderen Modells enthalten, das zusätzlich noch weitere Prädiktoren berücksichtigt (vgl. Abschn. 6.3.3). Solche Modellvergleiche können mit `anova(fitR, fitU, test="Chisq")` durchgeführt werden, wobei `fitR` das eingeschränkte und `fitU` das umfassendere Modell ist. Zusätzlich lässt sich wie in der linearen Regression `drop1(glm-Modell, test="Chi")` verwenden (vgl. Abschn. 6.3.3).

Um das Gesamtmodell mit einem Likelihood-Quotienten-Test auf Signifikanz zu prüfen, muss somit `anova(0-Modell, glm-Modell, test="Chisq")` aufgerufen werden (vgl. Abschn. 6.3.3). Der Test beruht auf dem Vergleich des angepassten Modells mit dem 0-Modell, das nur eine Konstante als Prädiktor beinhaltet. Teststatistik ist die Devianz-Differenz beider Modelle

¹²Bei Wald-Tests kann etwa das *Hauck-Donner-Phänomen* auftreten: Bei starken Effekten (sehr große β_j) sind die berechneten Streuungen $\hat{\sigma}_b$ dann deutlich zu groß, wodurch Wald-Tests der Parameter fälschlicherweise nicht signifikant werden.

mit der Differenz ihrer Freiheitsgrade als Anzahl der Freiheitsgrade der asymptotisch gültigen χ^2 -Verteilung.

```
> glm0 <- update(glmFit, . ~ 1) # 0-Modell
> anova(glm0, glmFit, test="Chisq")
Analysis of Deviance Table

Model 1: postFac ~ 1
Model 2: postFac ~ DVpre + IV
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1       29     41.455
2       26     24.409  3    17.047 0.0006912 ***

# manuelle Kontrolle
> chisqStat <- glmFit$null.deviance - deviance(glmFit)
> chisqDf   <- glmFit$df.null      - df.residual(glmFit)
> (pVal     <- pchisq(chisqStat, chisqDf, lower.tail=FALSE))
[1] 0.0006912397
```

Da der hier im Modell berücksichtigte Faktor IV mit mehreren Parametern β_j assoziiert ist, muss seine Signifikanz insgesamt über einen Modellvergleich gegen das vollständige Modell getestet werden.

```
# eingeschränktes Modell ohne Faktor IV
> glmPre <- update(glmFit, . ~ . - IV) # Modellvergleich
> anova(glmPre, glmFit, test="Chisq")
Analysis of Deviance Table

Model 1: postFac ~ DVpre
Model 2: postFac ~ DVpre + IV
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1       28     26.566
2       26     24.409  2    2.1572  0.3401
```

Analog erfolgt für Quadratsummen vom Typ I (vgl. Abschn. 7.5.2) der Test des kontinuierlichen Prädiktors DVpre als Vergleich des Modells nur mit DVpre mit dem 0-Modell.

```
> anova(glm0, glmPre, test="Chisq") # Modellvergleich
Analysis of Deviance Table

Model 1: postFac ~ 1
Model 2: postFac ~ DVpre
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1       29     41.455
2       28     26.566  1    14.89 0.000114 ***
```

8.1.6 Andere Link-Funktionen

Eine meist zu ähnlichen Ergebnissen führende Alternative zur logistischen Regression ist die Probit-Regression. Sie verwendet $\Phi^{-1}(\cdot)$, die Umkehrfunktion der Verteilungsfunktion $\Phi(\cdot)$ der Standardnormalverteilung als Link-Funktion, wofür das Argument `family` von `glm()` auf `binomial(link="probit")` zu setzen ist. Hier ist also $P = \Phi(\mathbf{X}\boldsymbol{\beta})$. Eine weitere Link-Funktion, die mit bedingter Binomialverteilung von Y kombiniert werden kann, ist die komplementäre log-log-Funktion $\ln(-\ln(1 - P))$. Man erhält sie mit `binomial(link="cloglog")`. Mit ihrer Umkehrfunktion, der Verteilungsfunktion der Gumbel-Verteilung, gilt $P = 1 - \exp(-\exp(\mathbf{X}\boldsymbol{\beta}))$ – dabei steht $\exp(\cdot)$ aus typografischen Gründen für $e^{(\cdot)}$.

Mitunter streuen empirische Residuen stärker, als dies bei bedingter Binomialverteilung mit der Varianz $n_i P (1 - P)$ zu erwarten wäre (*overdispersion*). Ein Hinweis auf overdispersion ist ein Verhältnis von Residual-Devianz zu Residual-Freiheitsgraden, das deutlich von 1 abweicht. Für diesen Fall kann ein Binomial-ähnliches Modell verwendet werden, das einem zusätzlichen, aus den Daten zu schätzenden Streuungsparameter ϕ besitzt, mit dem für die bedingte Varianz $\sigma^2 = \phi n_i P (1 - P)$ gilt: Hierfür ist `family` auf `quasibinomial(link="logit")` zu setzen. Die Parameterschätzungen b sind dann identisch zur logistischen Regression, die geschätzten Streuungen $\hat{\sigma}_b$ jedoch unterschiedlich, was zu anderen Ergebnissen der inferenzstatistischen Tests führt.

Da die bedingte Verteilung der Daten in der quasi-binomial Familie bis auf Erwartungswert und Varianz unspezifiziert bleibt, ist die Likelihood-Funktion der Daten für gegebene Parameter unbekannt. Trotzdem kann jedoch die IWLS-Methode Parameter schätzen (vgl. Abschn. 8.1.7), wobei dann alle likelihood-basierten Kennwerte wie AIC oder pseudo- R^2 nicht zur Verfügung stehen. Durch die Schätzung des Streuungs-Parameters sind die Wald-Teststatistiken keine z -, sondern t -Werte. Analog sollten Modellvergleich mit einem F -Test über das Argument `anova(..., test="F")` durchgeführt werden statt über einen χ^2 -Test wie bei bekannter Varianz.

Den Spezialfall einer linearen Regression erhält man mit `gaussian(link="identity")`. Die Maximum-Likelihood-Schätzungen der Parameter stimmen dann mit den Schätzern der linearen Regression überein.

8.1.7 Mögliche Probleme bei der Modellanpassung

Die Maximum-Likelihood-Schätzung der Parameter ist nicht in geschlossener Form darstellbar, sondern muss mit einem numerischen Optimierungsverfahren gefunden werden – typischerweise über die IWLS-Methode (iterative weighted least squares). Diese numerische Suche nach dem Maximum der Likelihood-Funktion kann in seltenen Fällen fehlschlagen, auch wenn es ein eindeutiges Maximum gibt. Ursache einer nicht konvergierenden Suche ist häufig, dass sie an einem Punkt von Schätzungen beginnt, der zu weit vom tatsächlichen Maximum entfernt ist.

Konvergenz-Probleme können durch Warnmeldungen angezeigt werden (etwa *algorithm did not converge*), sich in sehr großen Schätzungen bei gleichzeitig sehr großen Standardfehlern äußern oder in einer Residual-Devianz, die größer als die Devianz des Null-Modells ist. In diesen Fällen ist das Ergebnis von `glm()` nicht gültig. Ob Konvergenz erreicht wurde, speichert die von `glm()` zurückgegebene Liste in der Komponente `converged`.

Über das Argument `start` lassen sich manuell Start-Werte für alle Parameter vorgeben, um die Konvergenz der Suche zu begünstigen. Eine Strategie für die Wahl guter Start-Werte besteht darin, eine gewöhnliche lineare Regression der logit-transformierten Variable Y durchzuführen und deren Parameter-Schätzungen zu wählen. Konvergenzprobleme lassen sich u.U. auch über eine höhere maximale Anzahl von Suchschritten beheben, die in der Voreinstellung 25 beträgt und so geändert werden kann:

```
> glm(..., control=glm.control(maxit=<Anzahl>))
```

Die Maximum-Likelihood-Schätzung der Parameter setzt voraus, dass keine (quasi-) vollständige Separierbarkeit von Prädiktoren durch die Kategorien von Y vorliegt. Dies ist etwa der Fall, wenn der gesamte obere Wertebereich eines Prädiktors j nur im Zusammenhang mit Treffern auftritt und Y so stellenweise perfekt aus den Daten vorhersagbar ist. Das zugehörige geschätzte odds ratio $e^{\hat{\beta}_j}$ müsste dann ∞ sein.

Ein Symptom für (quasi-) vollständige Separierbarkeit ist die Warnmeldung, dass geschätzte Trefferwahrscheinlichkeiten von 0 bzw. 1 aufgetreten sind. Ein weiteres Symptom für Separierbarkeit sind sehr große Parameterschätzungen, die mit großen Standardfehlern assoziiert sind und daher im Wald-Test kleine z -Werte liefern. In diesem Fall kann auf penalisierte Verfahren ausgewichen werden, etwa auf die logistische Regression mit Firth-Korrektur, die im Paket `logistf` (Heinze, Ploner, Dunkler & Southworth, 2013) umgesetzt wird (vgl. auch Abschn. 6.6.2).

8.2 Ordinale Regression

In der ordinalen Regression soll eine kategoriale Variable Y mit k geordneten Kategorien $1, \dots, g, \dots, k$ mit p Prädiktoren X_j vorhergesagt werden. Dazu führt man die Situation auf jene der logistischen Regression zurück (vgl. Abschn. 8.1), indem man zunächst $k - 1$ dichotome Kategorisierungen $Y \geq g$ vs. $Y < g$ mit $g = 2, \dots, k$ vornimmt. Mit diesen dichotomen Kategorisierungen lassen sich nun $k - 1$ *kumulative Logits* bilden:

$$\text{logit}(P(Y \geq g)) = \ln \frac{P(Y \geq g)}{1 - P(Y \geq g)} = \ln \frac{P(Y = g) + \dots + P(Y = k)}{P(Y = 1) + \dots + P(Y = g - 1)}$$

Die $k - 1$ kumulativen Logits geben jeweils die logarithmierte Chance dafür an, dass Y mindestens die Kategorie g erreicht. Sie werden in $k - 1$ separaten logistischen Regressionen mit dem Modell $\text{logit}(P(Y \geq g)) = \beta_{0g} + \beta_1 X_1 + \dots + \beta_p X_p$ linear vorhergesagt ($g = 2, \dots, k$). Die Parameterschätzungen erfolgen dabei für alle Regressionen simultan mit der Nebenbedingung, dass die Menge der β_j für alle g identisch ist und $\beta_{02} > \dots > \beta_{0k}$ gilt. In diesem Modell führt ein höherer Prädiktorwert X_j bei positivem β_j zu einer höheren Chance, dass eine höhere Kategorie von Y erreicht wird.¹³

¹³Andere Formulierungen des Modells sind möglich. So legt etwa SPSS das Modell $\text{logit}(P(Y \leq g)) = \beta_{0g} - (\beta_1 X_1 + \dots + \beta_p X_p)$ mit der Nebenbedingung $\beta_{01} < \dots < \beta_{0k-1}$ zugrunde, das jedoch nur zu umgedrehten Vorzeichen der Schätzungen für die β_{0g} führt. Mit derselben Nebenbedingung ließe sich das Modell auch als $\text{logit}(P(Y \leq g)) = \beta_{0g} + \beta_1 X_1 + \dots + \beta_p X_p$ formulieren. In diesem Modell führt ein höherer Prädiktorwert X_j bei positivem β_j zu einer höheren Chance, dass eine *niedrigere* Kategorie von Y erreicht wird. Entsprechend haben hier die Schätzungen für alle Parameter umgekehrte Vorzeichen.

Kern des Modells ist die Annahme, dass eine additive Erhöhung des Prädiktorwerts X_j um den Wert c dazu führt, dass die Chance für eine höhere Kategorie unabhängig von g um den festen Faktor $e^{\beta_j c}$ wächst: $\frac{P(Y \geq g|X_j+c)}{P(Y < g|X_j+c)} = e^{\beta_j c} \frac{P(Y \geq g|X_j)}{P(Y < g|X_j)}$. Aus diesem Grund heißt es auch *proportional odds* Modell. Wie in der logistischen Regression ist damit e^{β_j} das odds ratio, also der Faktor, um den der vorhergesagte Wettquotient zunimmt, wenn X_j um eine Einheit wächst.¹⁴

Wegen $P(Y = g) = P(Y \geq g) - P(Y \geq g+1)$ hat die Link-Funktion sowie ihre Umkehrfunktion hier eine zusammengesetzte Form. Mit der logistischen Funktion als Umkehrfunktion der Logit-Funktion sind die theoretischen Parameter $P(Y = g)$ identifizierbar als:

$$P(Y = g) = \frac{e^{\beta_0 g + \beta_1 X_1 + \dots + \beta_p X_p}}{1 + e^{\beta_0 g + \beta_1 X_1 + \dots + \beta_p X_p}} - \frac{e^{\beta_0 g+1 + \beta_1 X_1 + \dots + \beta_p X_p}}{1 + e^{\beta_0 g+1 + \beta_1 X_1 + \dots + \beta_p X_p}}$$

Die Modellannahmen lassen sich grafisch veranschaulichen (Abb. 8.3): Bei einem Prädiktor X sind die $k - 1$ kumulativen Logits lineare Funktionen von X mit derselben Steigung und geordneten y -Achsenabschnitten. Die Wahrscheinlichkeit dafür, dass Y mindestens die Kategorie $g = 2, \dots, k$ erreicht, ergibt sich aus $k - 1$ parallelen logistischen Funktionen mit der horizontalen Verschiebung $\frac{\beta_{0,g+1} - \beta_{0,g}}{\beta_1}$, d. h. $P(Y \geq g+1|X) = P(Y \geq g|X - \frac{\beta_{0,g+1} - \beta_{0,g}}{\beta_1})$.

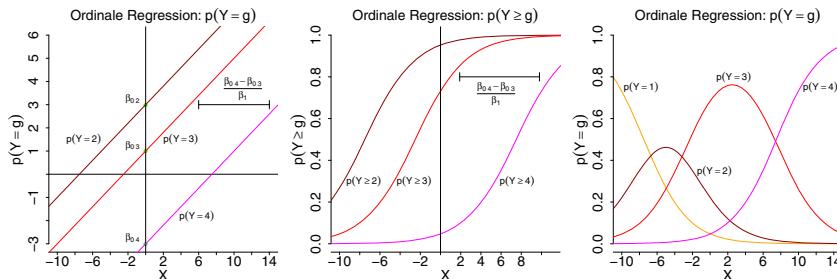


Abbildung 8.3: Modellannahmen der ordinalen Regression bei einem Prädiktor und 4 Kategorien des Kriteriums Y : Linearität der 3 kumulativen Logits mit identischer Steigung; parallel verschobene logistische Funktionen jeweils für die Wahrscheinlichkeit, mindestens Kategorie g zu erreichen; stochastisch geordnete Funktionen für die Wahrscheinlichkeit von $Y = g$.

8.2.1 Modellanpassung

Das proportional odds Modell mit kumulativen Logits kann mit `vglm()` aus dem Paket VGAM (Yee, 2010) angepasst werden. Diese Funktion hat gegenüber anderen, später ebenfalls erwähnten Funktionen den Vorteil, dass sie sich für alle in diesem Kapitel behandelten Modelle eignet und damit eine konsistente Herangehensweise an verwandte Fragestellungen ermöglicht.

```
> vglm(<Modellformel>, family=<Modell>, data=<Datensatz>)
```

¹⁴ Alternative proportional odds Modelle sind zum einen mit *adjacent category* Logits $\ln \frac{P(Y=g)}{P(Y=g-1)}$ möglich, zum anderen mit *continuation ratio (sequentiellen)* Logits $\ln \frac{P(Y=g)}{P(Y < g)}$.

Die Modellformel ist wie bei `glm()` zu formulieren, ihre linke Seite muss ein geordneter Faktor sein (vgl. Abschn. 2.6.4). Für das proportional odds Modell ist `family=propodds` zu setzen.¹⁵ Schließlich benötigt `data` den Datensatz mit den Variablen aus der Modellformel.

Als Beispiel soll eine Variable mit $k = 4$ geordneten Kategorien anhand von $p = 2$ Prädiktoren vorhergesagt werden. Die kategoriale AV soll sich dabei aus der Diskretisierung einer kontinuierlichen Variable ergeben.

```
> N      <- 100
> X1    <- rnorm(N, 175, 7)                      # Prädiktor 1
> X2    <- rnorm(N, 30, 8)                        # Prädiktor 2
> Ycont <- 0.5*X1 - 0.3*X2 + 10 + rnorm(N, 0, 6) # kontinuierliche AV

# geordneter Faktor aus Diskretisierung der kontinuierlichen AV
> Yord <- cut(Ycont, breaks=quantile(Ycont), include.lowest=TRUE,
+               labels=c("--", "-", "+", "++"), ordered=TRUE)

# zugehöriger ungeordneter Faktor für spätere Auswertungen
> Ycateg <- factor(Yord, ordered=FALSE)
> dfOrd  <- data.frame(X1, X2, Yord, Ycateg)      # Datensatz

> library(VGAM)                                    # für vglm(), lrtest()
> vglmFit <- vglm(Yord ~ X1 + X2, family=propodds, data=dfOrd)
```

Die $k - 1$ Schätzungen der Parameter β_{0j} sowie die p Schätzungen der Parameter β_j können mit `coef(vglm-Modell)` extrahiert werden. Die b_j sind wie in der logistischen Regression zu exponentiiert, um die geschätzten odds ratios zu erhalten (vgl. Abschn. 8.1.1): e^{b_j} gibt an, um welchen Faktor die vorhergesagte Chance wächst, eine höhere Kategorie zu erreichen, wenn sich X_j um eine Einheit erhöht. Die proportional odds Annahme bedeutet, dass sich die Wechselchance bei allen Kategorien im selben Maße ändert.

```
> exp(coef(vglmFit))                            # odds ratios
(Intercept):1 (Intercept):2 (Intercept):3          X1           X2
3.554552e-11 9.225479e-12 2.195724e-12 1.170903e+00 9.355142e-01
```

8.2.2 Anpassungsgüte

Wie in der logistischen Regression dienen verschiedene Maße als Anhaltspunkte zur Anpassungsgüte des Modells (vgl. Abschn. 8.1.3). Dazu zählt die Devianz ebenso wie das Informationskriterium AIC sowie die pseudo- R^2 Kennwerte. Letztere basieren auf dem Vergleich der geschätzten Likelihoods von 0-Modell und vollständigem Modell und sind manuell zu ermitteln. Das 0-Modell ist dabei das Regressionsmodell ohne Prädiktoren X_j .¹⁶

¹⁵Mit `vglm()` ist es möglich, auch die proportional odds Modelle mit adjacent category Logits bzw. continuation ratio Logits anzupassen (vgl. Abschn. 8.2, Fußnote 14). Dazu ist `family` auf `acat(parallel=TRUE)` bzw. auf `sratio(parallel=TRUE)` zu setzen. Eine weitere Option für `acat()` bzw. `sratio()` ist dabei das Argument `reverse`, das die Vergleichsrichtung dieser Logits bzgl. der Stufen von Y kontrolliert und auf `TRUE` oder `FALSE` gesetzt werden kann.

¹⁶Weitere Gütemaße der Modellanpassung erzeugt `lrm()` aus dem Paket `rms` (vgl. Abschn. 8.1.3, Fußnote 9).

```

> deviance(vglmFit)                      # Devianz
[1] 244.1915

> AIC(vglmFit)                          # Informationskriterium AIC
[1] 254.1915

# O-Modell für pseudo-R^2 Kennwerte
> vglm0 <- vglm(Yord ~ 1, family=propodds, data=df0rd)
> LL0   <- logLik(vglm0)                # gesch. log-likelihood O-Modell
> LLf   <- logLik(vglmFit)              # gesch. log-likelihood vollst. Modell

> as.vector( 1 - (LLf / LL0))           # R^2 McFadden
[1] 0.1192653

> as.vector( 1 - exp((2/N) * (LL0 - LLf)))          # R^2 Cox & Snell
[1] 0.2815605

# R^2 Nagelkerke
> as.vector((1 - exp((2/N) * (LL0 - LLf))) / (1 - exp(LL0^(2/N))))
[1] 0.3003312

```

Für potentiell auftretende Probleme bei der Modellanpassung vgl. Abschn. 8.1.7.

8.2.3 Signifikanztests für Parameter und Modell

Mit `summary(<vglm-Modell>)` werden neben den Parameterschätzungen b auch ihre geschätzten Streuungen $\hat{\sigma}_b$ sowie die zugehörigen z -Werte $\frac{b}{\hat{\sigma}_b}$ berechnet. Mit der Annahme, dass z unter der Nullhypothese asymptotisch standardnormalverteilt ist, stehen die zweiseitigen p -Werte in der Spalte `Pr(>|z|)`. Die genannten Werte lassen sich mit `coef(summary(<vglm-Modell>))` extrahieren.

```

> sumOrd  <- summary(vglmFit)
> (coefOrd <- coef(sumOrd))
      Estimate Std. Error    z value  Pr(>|z|)
(Intercept):1 -24.06020695 5.82571121 -4.130003 3.627579e-05
(Intercept):2 -25.40905205 5.87912090 -4.321914 1.546818e-05
(Intercept):3 -26.84450928 5.94234461 -4.517495 6.257564e-06
X1            0.15777487 0.03385696  4.660042 3.161443e-06
X2           -0.06665895 0.02499298 -2.667107 7.650735e-03

```

Ist von asymptotischer Normalverteilung der Schätzungen b_{j_g} auszugehen, können approximative $(1 - \alpha)$ -Wald-Konfidenzintervalle $[b - z_{1-\alpha/2} \hat{\sigma}_b, b + z_{\alpha/2} \hat{\sigma}_b]$ für die β_{j_g} berechnet werden, wobei $z_{\alpha/2}$ das $\frac{\alpha}{2}$ -Quantil der Standardnormalverteilung ist.¹⁷

¹⁷Für Konfidenzintervalle der Parameter kann die ordinale Regression auch zunächst mit `polr()` aus dem Paket `MASS` angepasst werden. Die dann von `confint(<polr-Modell>)` erzeugten Konfidenzintervalle basieren auf der Profile-Likelihood-Methode.

```
# 1-alpha/2 alpha/2 Quantile der Standardnormalverteilung
> zCrit <- qnorm(c(1 - 0.05/2, 0.05/2))

# Wald-Konfidenzintervalle für die Parameter (ohne intercepts b_0g)
> (ciCoef <- t(apply(coefOrd[-(1:3), ], 1, function(x) {
+           x["Estimate"] - zCrit*x["Std. Error"] } )))

[,1]      [,2]
X1  0.09141645  0.22413329
X2 -0.11564429 -0.01767361
```

Eine oft geeignetere Alternative zu Wald-Tests sind Likelihood-Quotienten-Tests eines umfassenderen Modells `fitU` gegen ein eingeschränktes Modell `fitR` mit `lrtest(fitU, fitR)` aus dem Paket `VGAM` (vgl. Abschn. 8.1.5).

```
# eingeschränktes Modell ohne Prädiktor X2
> vglmR <- vglm(Yord ~ X1, family=propodds, data=dfOrd)
> lrtest(vglmFit, vglmR)                                # Likelihood-Quotienten-Test
Likelihood ratio test

Model 1: Yord ~ X1 + X2
Model 2: Yord ~ X1
#Df LogLik Df Chisq Pr(>Chisq)
1 295 -122.10
2 296 -125.87  1 7.5568   0.005978 **
```

Auf diese Weise lässt sich auch das Gesamtmodell gegen das 0-Modell ohne Prädiktoren X_j testen.

```
# lrtest(vglmFit, vglm0)                                # Test des Gesamtmodells
Likelihood ratio test

Model 1: Yord ~ X1 + X2
Model 2: Yord ~ 1
#Df LogLik Df Chisq Pr(>Chisq)
1 295 -122.10
2 297 -138.63  2 33.067   6.6e-08 ***
```

Ohne die proportional odds Annahme, dass die Menge der β_j in allen $k - 1$ separaten Regressionen der kumulativen Logits identisch ist, erhält man ein umfassenderes Modell. Die Parameter β_{jg} sind dann von g abhängig, die linearen Funktionen für die Logits also nicht mehr parallel (Abb. 8.3). Dieses Modell lässt sich ebenfalls mit `vglm()` anpassen, indem man `family` auf `cumulative(parallel=FALSE)` setzt. Der Likelihood-Quotienten-Test des umfassenderen Modells gegen das eingeschränkte Modell mit proportional odds Annahme erfolgt wieder mit `lrtest()`. Fällt der Test signifikant aus, ist dies ein Hinweis darauf, dass die Daten gegen die proportional odds Annahme sprechen.

```
# eingeschränktes Modell - äquivalent zu vglm(..., family=propodds)
> vglmP <- vglm(Yord ~ X1 + X2, family=cumulative(parallel=TRUE,
+ reverse=TRUE), data=dfOrd)
```

```
# umfassenderes Modell ohne proportional odds Annahme
> vglmNP <- vglm(Yord ~ X1 + X2, family=cumulative(parallel=FALSE,
+                   reverse=TRUE), data=dfOrd)

> lrtest(vglmNP, vglmNP)                      # Likelihood-Quotienten-Test
Likelihood ratio test

Model 1: Yord ~ X1 + X2
Model 2: Yord ~ X1 + X2
#Df LogLik Df Chisq Pr(>Chisq)
1 291 -119.89
2 295 -122.10 4 4.4033 0.35422
```

8.2.4 Vorhersage, Klassifikation und Anwendung auf neue Daten

Die vorhergesagten Kategorienwahrscheinlichkeiten $\hat{P}(Y = g)$ erhält man wie in der logistischen Regression mit `predict(vglm-Modell, type="response")`. Die ausgegebene Matrix enthält für jede Beobachtung (Zeilen) die vorhergesagte Wahrscheinlichkeit für jede Kategorie (Spalten).

```
> PhatCateg <- predict(vglmFit, type="response")
> head(PhatCateg, n=3)
   Yord==--    Yord=-     Yord=+    Yord==++
1 0.1433775 0.2486796 0.3383702 0.26957271
2 0.2785688 0.3194630 0.2640555 0.13791266
3 0.5278107 0.2837526 0.1360684 0.05236823
```

Die vorhergesagten Kategorien selbst lassen sich aus den vorhergesagten Kategorienwahrscheinlichkeiten bestimmen, indem pro Beobachtung die Kategorie mit der maximalen vorhergesagten Wahrscheinlichkeit herangezogen wird. Das zeilenweise Maximum einer Matrix gibt `max.col(Matrix)` aus.

```
> categHat <- levels(dfOrd$Yord)[max.col(PhatCateg)]
> head(categHat)
[1] "+" "-" "--" "+" "--" "+"
```

Die Kontingenztafel tatsächlicher und vorhergesagter Kategorien eignet sich als Grundlage für die Berechnung von Übereinstimmungsmaßen wie der Rate der korrekten Klassifikation. Hier ist darauf zu achten, dass die Kategorien identisch geordnet sind.

```
# Faktor mit gleich geordneten Kategorien
> facHat <- factor(categHat, levels=levels(dfOrd$Yord))

# Kontingenztafel tatsächlicher und vorhergesagter Kategorien
> cTab <- table(dfOrd$Yord, facHat, dnn=c("Yord", "facHat"))
> addmargins(cTab)                                # Randsummen hinzufügen
```

```

facHat
Yord   --   -   +   ++ Sum
--    10   7   7   1  25
-     10   7   4   4  25
+     5   7   7   6  25
++    0   2   9  14 25
Sum   25  23  27  25 100

> (CCR <- sum(diag(cTab)) / sum(cTab)) # Rate der korrekten Klass.
[1] 0.38

An das Argument newdata von predict() kann zusätzlich ein Datensatz übergeben werden, der neue Daten für Variablen mit denselben Namen, und bei Faktoren zusätzlich denselben Stufen wie jene der ursprünglichen Prädiktoren enthält. Als Ergebnis erhält man die vorhergesagten Kategorienwahrscheinlichkeiten für die neuen Prädiktorwerte (vgl. Abschn. 6.4).

> Nnew <- 3
> dfNew <- data.frame(X1=rnorm(Nnew, 175, 7),           # neue Daten
+                         X2=rnorm(Nnew, 30, 8))

> predict(vglmFit, newdata=dfNew, type="response")      # Vorhersage
   Yord==--   Yord=-   Yord=+   Yord=++
1 0.10563202 0.2071139 0.3438480 0.3434060
2 0.17385833 0.2739122 0.3253090 0.2269204
3 0.31326643 0.3240997 0.2433696 0.1192643

```

8.3 Multinomiale Regression

In der multinomialen Regression soll eine kategoriale Variable Y mit k ungeordneten Kategorien $1, \dots, g, \dots, k$ mit p Prädiktoren X_j vorhergesagt werden. Dazu führt man die Situation auf jene der logistischen Regression zurück (vgl. Abschn. 8.1), indem man zunächst eine Referenzkategorie r von Y festlegt und $k-1$ Logits der Form $\ln \frac{P(Y=g)}{P(Y=r)}$ bildet. Diese $k-1$ *baseline category Logits* geben die logarithmierte Chance dafür an, dass Y die Kategorie g annimmt – verglichen mit der Referenzkategorie r . Für r wird in der Voreinstellung typischerweise die erste (so hier im folgenden) oder die letzte Kategorie von Y gewählt.

Die *baseline category Logits* werden in $k-1$ separaten logistischen Regressionen mit dem Modell $\ln \frac{P(Y=g)}{P(Y=1)} = \beta_{0g} + \beta_{1g}X_1 + \dots + \beta_{pg}X_p$ linear vorhergesagt ($g = 1, \dots, k$).¹⁸ Die Parameterschätzungen erfolgen für alle Regressionen simultan, wobei anders als in der ordinalen Regression sowohl die β_{0g} als auch die Gewichte β_{jg} als von g abhängig betrachtet werden (Abb. 8.4). In diesem Modell führt ein höherer Prädiktorwert X_j bei positivem β_j zu einer höheren

¹⁸Kurz $\ln \frac{P(Y=g)}{P(Y=1)} = \mathbf{X}\boldsymbol{\beta}_g$. In der Referenzkategorie 1 sind die Parameter wegen $\ln \frac{P(Y=1)}{P(Y=1)} = \ln 1 = 0$ festgelegt, und es gilt $\beta_{01} = \beta_{j1} = 0$ (mit $j = 1, \dots, p$) sowie $e^{\mathbf{X}\boldsymbol{\beta}_g} = e^0 = 1$.

Chance, dass die Kategorie g von Y angenommen wird – verglichen mit der Referenzkategorie.¹⁹ Mit den gewählten baseline category Logits sind auch alle verbleibenden logarithmierten Chancen beim Vergleich von je zwei Kategorien a, b von Y festgelegt:

$$\begin{aligned}\ln \frac{P(Y=a)}{P(Y=b)} &= \ln \frac{P(Y=a)/P(Y=1)}{P(Y=b)/P(Y=1)} = \ln \frac{P(Y=a)}{P(Y=1)} - \ln \frac{P(Y=b)}{P(Y=1)} \\ &= \beta_{0_a} + \beta_{1_a} X_1 + \cdots + \beta_{p_a} X_p \\ &\quad - \beta_{0_b} + \beta_{1_b} X_1 + \cdots + \beta_{p_b} X_p \\ &= (\beta_{0_a} - \beta_{0_b}) + (\beta_{1_a} - \beta_{1_b}) X_1 + \cdots + (\beta_{p_a} - \beta_{p_b}) X_p\end{aligned}$$

Mit der logistischen Funktion als Umkehrfunktion der Logit-Funktion sind die theoretischen Parameter $P(Y = g)$ identifizierbar als:

$$P(Y = g) = \frac{e^{\beta_{0_g} + \beta_{1_g} X_1 + \cdots + \beta_{p_g} X_p}}{\sum_{c=1}^k e^{\beta_{0_c} + \beta_{1_c} X_1 + \cdots + \beta_{p_c} X_p}} = \frac{e^{\beta_{0_g} + \beta_{1_g} X_1 + \cdots + \beta_{p_g} X_p}}{1 + \sum_{c=2}^k e^{\beta_{0_c} + \beta_{1_c} X_1 + \cdots + \beta_{p_c} X_p}}$$

Insbesondere bestimmt sich die Wahrscheinlichkeit der Referenzkategorie 1 als:

$$P(Y = 1) = \frac{1}{1 + \sum_{c=2}^k e^{\beta_{0_c} + \beta_{1_c} X_1 + \cdots + \beta_{p_c} X_p}}$$

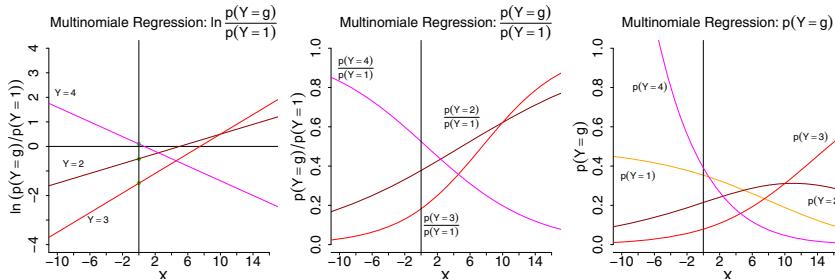


Abbildung 8.4: Multinomiale Regression mit einem Prädiktor und 4 Kategorien der AV Y : Linearität der 3 baseline category Logits mit unterschiedlichen Steigungen; logistische Funktionen für die Chance, verglichen mit der baseline Kategorie eine Kategorie g zu erhalten; zugehörige Funktionen für die Wahrscheinlichkeit einer Kategorie g von Y .

8.3.1 Modellanpassung

Als Beispiel sollen die Daten der ordinalen Regression in Abschn. 8.2.1 herangezogen werden, wobei als AV nun der dort bereits erstellte ungeordnete Faktor dient. Die Anpassung des

¹⁹Dabei wird *Unabhängigkeit von irrelevanten Alternativen* angenommen: Für die Chance beim paarweisen Vergleich von g mit der Referenzkategorie soll die Existenz weiterer Kategorien irrelevant sein. Ohne diese Annahme kommen etwa Bradley-Terry-Modelle aus, von denen eine eingeschränkte Variante mit `brat()` aus dem Paket **VGAM** angepasst werden kann.

Modells erfolgt wieder mit `vglm()` aus dem Paket VGAM. Dafür ist das Argument `family` auf `multinomial(refLevel=1)` zu setzen, womit gleichzeitig die Referenzkategorie `refLevel` auf die erste Stufe von Y festgelegt werden kann. Wie in der logistischen Regression sind die Schätzungen b_{jg} zu exponentiiert, um die geschätzten odds ratios zu erhalten: $e^{b_{jg}}$ gibt bezogen auf die Referenzkategorie an, um welchen Faktor die vorhergesagte Chance wächst, Kategorie g zu erreichen, wenn sich X_j um eine Einheit erhöht (vgl. Abschn. 8.1.1).

```
> library(VGAM)                                # für vglm(), lrtest()
> vglmFitMN <- vglm(Ycateg ~ X1 + X2, family=multinomial(refLevel=1),
+                      data=df0rd)

> exp(coef(vglmFitMN))                         # odds ratios
(Intercept):1 (Intercept):2 (Intercept):3          X1:1           X1:2
5.812237e-02  1.579953e-11  8.460664e-20  1.019506e+00  1.162001e+00
X1:3          X2:1          X2:2          X2:3
1.316564e+00  9.854338e-01  9.636944e-01  8.647246e-01
```

8.3.2 Anpassungsgüte

Wie in der logistischen Regression dienen verschiedene Maße als Anhaltspunkte zur Anpassungsgüte des Modells (vgl. Abschn. 8.1.3). Dazu zählt die Devianz ebenso wie das Informationskriterium AIC sowie die pseudo- R^2 Kennwerte. Letztere basieren auf dem Vergleich der geschätzten Likelihoods von 0-Modell und vollständigem Modell und sind manuell zu ermitteln. Das 0-Modell ist dabei das Regressionsmodell ohne Prädiktoren X_j .

```
> deviance(vglmFitMN)                         # Devianz
[1] 237.148

> AIC(vglmFitMN)                             # Informationskriterium AIC
[1] 255.148

# 0-Modell für pseudo-R^2 Kennwerte
> vglmMNO <- vglm(Ycateg ~ 1, family=multinomial, data=df0rd)
> LL0     <- logLik(vglmMNO)      # gesch. log-likelihood 0-Modell
> LLf     <- logLik(vglmFitMN)    # gesch. log-likelihood vollst. Modell

> as.vector( 1 - (LLf / LL0))                # R^2 McFadden
[1] 0.1446692

> as.vector( 1 - exp((2/N) * (LL0 - LLf)))   # R^2 Cox & Snell
[1] 0.3304224

# R^2 Nagelkerke
> as.vector((1 - exp((2/N) * (LL0 - LLf))) / (1 - exp(LL0^(2/N))))
[1] 0.3524506
```

Für potentiell auftretende Probleme bei der Modellanpassung vgl. Abschn. 8.1.7.

8.3.3 Signifikanztests für Parameter und Modell

Mit `summary(⟨vglm-Modell⟩)` werden neben den Parameterschätzungen b auch ihre geschätzten Streuungen $\hat{\sigma}_b$ sowie die zugehörigen z -Werte $\frac{b}{\hat{\sigma}_b}$ berechnet. Mit der Annahme, dass z unter der Nullhypothese asymptotisch standardnormalverteilt ist, stehen die zweiseitigen p -Werte in der Spalte `Pr(>|z|)`. Die genannten Werte lassen sich mit `coef(summary(⟨vglm-Modell⟩))` extrahieren.

```
> sumMN  <- summary(vglmFitMN)
> (coefMN <- coef(sumMN))
   Estimate Std. Error    z value Pr(>|z|)
(Intercept):1 -2.84520460 8.86963583 -0.3207803 7.483769e-01
(Intercept):2 -24.87104089 9.67909358 -2.5695630 1.018269e-02
(Intercept):3 -43.91627419 11.63358071 -3.7749576 1.600349e-04
X1:1          0.01931815 0.05232332 0.3692072 7.119733e-01
X1:2          0.15014353 0.05662132 2.6517138 8.008439e-03
X1:3          0.27502499 0.06780602 4.0560557 4.990836e-05
X2:1          -0.01467332 0.03810166 -0.3851098 7.001561e-01
X2:2          -0.03698110 0.04095569 -0.9029539 3.665504e-01
X2:3          -0.14534420 0.04957956 -2.9315347 3.372917e-03
```

Ist von asymptotischer Normalverteilung der Schätzungen b_{j_g} auszugehen, können approximative $(1 - \alpha)$ -Wald-Konfidenzintervalle $[b - z_{1-\alpha/2} \hat{\sigma}_b, b + z_{\alpha/2} \hat{\sigma}_b]$ für die β_{j_g} berechnet werden, wobei $z_{\alpha/2}$ das $\frac{\alpha}{2}$ -Quantil der Standardnormalverteilung ist.

```
# 1-alpha/2 und alpha/2 Quantile der Standardnormalverteilung
> zCrit <- qnorm(c(1 - 0.05/2, 0.05/2))

# Wald-Konfidenzintervalle für die Parameter (ohne intercepts b_0g)
> (ciCoef <- t(apply(coefMN[-(1:3), ], 1, function(x) {
+   x[["Estimate"]] - zCrit*x[["Std. Error"]] } )))
      [,1]      [,2]
X1:1 -0.08323367 0.12186997
X1:2  0.03916779 0.26111927
X1:3  0.14212763 0.40792234
X2:1 -0.08935120 0.06000456
X2:2 -0.11725276 0.04329057
X2:3 -0.24251836 -0.04817005
```

Da jeder Prädiktor mit mehreren Parametern β_{j_g} assoziiert ist, müssen Prädiktoren selbst über Modellvergleiche auf Signifikanz getestet werden (vgl. Abschn. 8.1.5). Dazu dienen Likelihood-Quotienten-Tests, die auf der asymptotisch χ^2 -verteilten Devianz-Differenz zweier nested Modelle mit demselben Kriterium beruhen: Der Prädiktorensatz des eingeschränkten Modells `⟨fitR⟩` ist dabei vollständig im Prädiktorensatz des umfassenderen Modells `⟨fitU⟩` enthalten, das zusätzlich noch weitere Prädiktoren berücksichtigt. Der Test erfolgt dann mit `lrtest(⟨fitU⟩, ⟨fitR⟩)` aus dem Paket `VGAM`.

```
# eingeschränktes Modell ohne Prädiktor X2
```

```
> vglmFitR <- vglm(Ycateg ~ X1, family=multinomial(refLevel=1), data=dfOrd)
> lrtest(vglmFitMN, vglmFitR) # Modellvergleich
Likelihood ratio test

Model 1: Ycateg ~ X1 + X2
Model 2: Ycateg ~ X1
#Df LogLik Df Chisq Pr(>Chisq)
1 291 -118.57
2 294 -124.50 3 11.852 0.007906 **
```

8.3.4 Vorhersage, Klassifikation und Anwendung auf neue Daten

Die vorhergesagten Kategorienwahrscheinlichkeiten $\hat{P}(Y = g)$ erhält man wie in der logistischen Regression mit `predict(vglm-Modell, type="response")`. Die ausgegebene Matrix enthält für jede Beobachtung (Zeilen) die vorhergesagte Wahrscheinlichkeit für jede Kategorie (Spalten). Mit der gewählten Herangehensweise ist die mittlere vorhergesagte Wahrscheinlichkeit für jede Kategorie gleich der empirischen relativen Häufigkeit der Kategorie.²⁰

```
> PhatCateg <- predict(vglmFitMN, type="response")
> head(PhatCateg, n=3)
--      -      +      ++
1 0.1946834 0.2255730 0.2982816 0.28146203
2 0.3219964 0.3219154 0.2675170 0.08857120
3 0.4473732 0.3769135 0.1596527 0.01606071

# mittlere vorhergesagte Kategorien-Wahrscheinlichkeiten
> colMeans(PhatCateg)
--      -      +      ++
0.25 0.25 0.25 0.25

# empirische relative Kategorien-Häufigkeiten
> prop.table(table(dfOrd$Ycateg))
--      -      +      ++
0.25 0.25 0.25 0.25
```

Die vorhergesagten Kategorien selbst lassen sich aus den vorhergesagten Kategorienwahrscheinlichkeiten bestimmen, indem pro Beobachtung die Kategorie mit der maximalen vorhergesagten Wahrscheinlichkeit herangezogen wird. Das zeilenweise Maximum einer Matrix gibt `max.col(Matrix)` aus.

```
> categHat <- levels(dfOrd$Ycateg)[max.col(PhatCateg)]
> head(categHat)
[1] "+" "--" "---" "+" "--" "+"
```

²⁰Dies ist der Fall, wenn die kanonische Link-Funktion und Maximum-Likelihood-Schätzungen der Parameter gewählt werden und das Modell die absoluten Terme β_{0_g} besitzt.

Die Kontingenztafel tatsächlicher und vorhergesagter Kategorien eignet sich als Grundlage für die Berechnung von Übereinstimmungsmaßen wie der Rate der korrekten Klassifikation. Hier ist darauf zu achten, dass die Kategorien identisch geordnet sind.

```
> facHat <- factor(categHat, levels=levels(dfOrd$Ycateg))
> cTab   <- table(dfOrd$Ycateg, facHat, dnn=c("Ycateg", "facHat"))
> addmargins(cTab)                                # Randsummen hinzufügen
      facHat
Ycateg -- - + ++ Sum
--   9 5 8 3 25
-   11 5 5 4 25
+   5 5 8 7 25
++  1 2 8 14 25
Sum 26 17 29 28 100

> (CCR <- sum(diag(cTab)) / sum(cTab)) # Rate der korrekten Klass.
[1] 0.36
```

An das Argument `newdata` von `predict()` kann zusätzlich ein Datensatz übergeben werden, der neue Daten für Variablen mit denselben Namen, und bei Faktoren zusätzlich denselben Stufen wie jene der ursprünglichen Prädiktoren enthält. Als Ergebnis erhält man die vorhergesagten Kategorienwahrscheinlichkeiten für die neuen Prädiktorwerte (vgl. Abschn. 6.4).

```
> Nnew  <- 3
> dfNew <- data.frame(X1=rnorm(Nnew, 175, 7),           # neue Daten
+                      X2=rnorm(Nnew, 30, 8))

> predict(vglmFitMN, newdata=dfNew, type="response") # Vorhersage
      --      -      +      ++
1 0.05150391 0.06762012 0.2943623 0.58651367
2 0.45228458 0.39748171 0.1311896 0.01904412
3 0.15948745 0.18932511 0.3110646 0.34012280
```

8.4 Regression für Zähldaten

Das GLM bietet verschiedene Möglichkeiten zur Modellierung einer Variable Y , die nur ganzzahlige nichtnegative Werte annehmen kann und keine obere Schranke aufweist, wie es für Zähldaten charakteristisch ist. Die von Y gezählten Ereignisse sollen dabei unabhängig voneinander eintreten. Handelt es sich bei den Prädiktoren X_j um kontinuierliche Variablen, spricht man von Regressionsmodellen, bei der Modellierung einer festen Gesamtzahl von Ereignissen durch Gruppierungsfaktoren meist allgemein von log-linearen Modellen (vgl. Abschn. 8.5).

8.4.1 Poisson-Regression

Bei der Poisson-Regression wird als bedingte Verteilung von Y eine Poisson-Verteilung $P(Y = y) = \frac{\mu^y e^{-\mu}}{y!}$ angenommen, mit dem Erwartungswert $\mu = E(Y)$. Die kanonische Link-Funktion

ist der natürliche Logarithmus, das lineare Modell ist also $\ln \mu = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p = \mathbf{X}\boldsymbol{\beta}$. Die bedingten Verteilungen von Y sind dann bereits vollständig festgelegt, da ihre Varianz ebenfalls gleich μ ist. Der Erwartungswert ist als $\mu = e^{\mathbf{X}\boldsymbol{\beta}} = e^{\beta_0} e^{\beta_1 X_1} \dots e^{\beta_p X_p}$ identifizierbar. Einem exponenzierten Parameter e^{β_j} kommt deshalb die Bedeutung des multiplikativen Faktors zu, mit dem μ wächst, wenn sich der Prädiktor X_j um eine Einheit vergrößert.²¹

Die Anpassung einer Poisson-Regression geschieht wie in der logistischen Regression mit `glm()`, wobei das Argument `family` auf `poisson(link="log")` zu setzen ist (vgl. Abschn. 8.1.1).²² Die Daten der vorherzusagenden Variable müssen aus ganzzahligen nichtnegativen Werten bestehen.

Für ein Beispiel sollen zunächst Zähldaten simuliert werden, die mit zwei Prädiktoren korrelieren. Dazu wird die `rmvnorm()` Funktion des `mvtnorm` Pakets verwendet, die Zufallsvektoren einer multinormalverteilten Variable simuliert. Die Verwendung von `rmvnorm()` gleicht der von `rnorm()`, lediglich muss hier das theoretische Zentroid μ für das Argument `mean` und die theoretische Kovarianzmatrix Σ für `sigma` angegeben werden. Die Daten einer der erzeugten Variablen werden zusätzlich gerundet und ihre negativen Werte auf Null gesetzt, damit sie als Zählvariable dienen kann.

```
> library(mvtnorm)                                     # für rmvnorm()
> N          <- 200
> sigma      <- matrix(c(4,2,-3, 2,16,-1, -3,-1,8), byrow=TRUE, ncol=3)
> mu         <- c(-3, 2, 4)                         # Erwartungswerte
> XY         <- rmvnorm(N, mean=mu, sigma=sigma)    # Zufallsdaten
> Y          <- round(XY[, 3] - 1.5)                 # runde Zähldaten
> Y[Y < 0] <- 0                                     # negative Werte -> 0
> dfCount   <- data.frame(X1=XY[, 1], X2=XY[, 2], Y) # Datensatz

# Poisson-Regression
> glmFitP <- glm(Y ~ X1 + X2, family=poisson(link="log"), data=dfCount)
```

Wie in der logistischen Regression erhält man die Parameterschätzungen b_j mit `coef()` und die Konfidenzintervalle für die Parameter β_j mit `confint()` (vgl. Abschn. 8.1.1, Fußnote 4). Die geschätzten Änderungsfaktoren für $E(Y)$ ergeben sich durch Exponenzieren als e^{b_j} .

```
> exp(coef(glmFitP))                                # Änderungsfaktoren
(Intercept)           X1           X2
1.1555238  0.7692037  1.0205773

> exp(confint(glmFitP))                            # zugehörige Konfidenzintervalle
           2.5 %     97.5 %
(Intercept) 0.9479450 1.3993779
X1          0.7380968 0.8015816
X2          0.9992731 1.0424328
```

Wald-Tests der Parameter und Informationen zur Modellpassung insgesamt liefert `summary()` (vgl. Abschn. 8.1.3, 8.1.5).

²¹Für einen Prädiktor X : $\mu_{X+1} = e^{\beta_0 + \beta_1(X+1)} = e^{\beta_0} e^{\beta_1(X+1)} = e^{\beta_0} e^{\beta_1 X} e^{\beta_1} = e^{\beta_1} e^{\beta_0 + \beta_1 X} = e^{\beta_1} \mu_X$.

²²Bei der Verwendung von `vglm()` aus dem Paket `VGAM` ist das Argument `family` auf `poissonff` zu setzen.

```
> summary(glmFitP)                                # Parametertests + Modellpassung
Call:
glm(formula = Y ~ X1 + X2, family=poisson(link="log"), data=dfCount)

Deviance Residuals:
    Min      1Q  Median      3Q     Max 
-3.1695 -1.4608 -0.2707  0.7173  3.4757 

Coefficients:
            Estimate Std. Error z value Pr(>|z|)    
(Intercept) 0.14455   0.09935   1.455   0.146    
X1          -0.26240   0.02105 -12.466 <2e-16 ***  
X2          0.02037   0.01079   1.888   0.059 .    
                                                        
(Dispersion parameter for poisson family taken to be 1)

Null deviance: 536.93 on 199 degrees of freedom
Residual deviance: 374.78 on 197 degrees of freedom
AIC: 825.22

Number of Fisher Scoring iterations: 5
```

Wie in der logistischen Regression erhält man die vorhergesagten Häufigkeiten mit der Funktion `predict(<glm-Modell>, type="response")` (vgl. Abschn. 8.1.4). Über das Argument `newdata` lässt sich das angepasste Modell dabei auch auf neue Werte für dieselben Prädiktoren anwenden, um Vorhersagen zu gewinnen.

8.4.2 Ereignisraten analysieren

Die Poisson-Regression erlaubt es auch, Ereignisraten zu analysieren. Die absoluten Häufigkeiten Y sind dann auf eine bestimmte Referenzgröße t bezogen, die die Menge der potentiell beobachtbaren Ereignisse bestimmt. Bei t kann es sich etwa um die Länge eines Zeitintervalls handeln, in dem Ereignisse gezählt werden. In einer anderen Situation kann t die Größe einer Fläche sein, auf der die Anzahl von Elementen mit einer bestimmten Eigenschaft zu zählen sind, die jeweils einem Ereignis entsprechen. t wird als *exposure* bezeichnet und ist echt positiv.

Bezeichnet t die Zeittdauer, nimmt man an, dass der Abstand zwischen zwei aufeinander folgenden Ereignissen unabhängig von t exponentialverteilt mit Erwartungswert $\frac{1}{\lambda}$ ist. Dann folgt $E(Y) = \lambda t$ mit der Konstante λ als echt positiver Grundrate, mit der ein Ereignis pro Zeiteinheit auftritt.

Ist in einer Untersuchung die zu den beobachteten Häufigkeiten Y gehörende exposure t variabel, stellt die Grundrate $\lambda = \frac{\mu}{t}$ die zu modellierende Variable dar. Mit dem Logarithmus als Link-Funktion folgt $\mathbf{X}\beta = \ln \lambda = \ln \frac{\mu}{t} = \ln \mu - \ln t$, als lineares Vorhersagemodell ergibt sich $\ln \mu = \mathbf{X}\beta + \ln t$. Im linearen Prädiktor $\ln t + \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p$ ist $\ln t$ eine additive Konstante wie β_0 – allerdings mit dem Unterschied, dass t kein zu schätzender Parameter,

sondern durch die Daten festgelegt ist. $\ln t$ wird auch als *offset* bezeichnet. Die Grundrate λ ist identifizierbar als $\lambda = e^{X\beta + \ln t} = t e^{X\beta}$.

In der Anpassung der Poisson-Regression für Ereignisraten mit `glm()` ist das Argument `offset=log(<t>)` zu verwenden. Dabei ist `<t>` ein Vektor mit den Werten der exposure.

```
# simulierte Zähldaten, hier ohne Zusammenhang von Prädiktor und AV
> Nt    <- 100                                # Beobachtungsobjekte
> Ti    <- sample(20:40, Nt, replace=TRUE)      # exposure
> Xt    <- rnorm(Nt, 100, 15)                  # Prädiktor
> Yt    <- rbinom(Nt, size=Ti, prob=0.5)       # beobachtete Häufigkeiten
> fitT <- glm(Yt ~ Xt, family=poisson(link="log"), offset=log(Ti))
> summary(fitT)                               # ...
```

8.4.3 Adjustierte Poisson-Regression und negative Binomial-Regression

Oft streuen empirische Residuen deutlich stärker, als dies bei bedingter Poisson-Verteilung von Y zu erwarten wäre, bei der die Streuung gleich dem bedingten Erwartungswert ist (*overdispersion*). Ein Hinweis auf overdispersion ist ein Verhältnis von Residual-Devianz zu Residual-Freiheitsgraden, das deutlich von 1 abweicht. Die modellbasierten Streuungsschätzungen unterschätzen dann die wahre Streuung, was zu liberale Signifikanztests der Parameter zur Folge hat. Eine mögliche Ursache für overdispersion ist ein unvollständiges Vorhersagemodell, das tatsächlich relevante Prädiktoren nicht berücksichtigt. Bei overdispersion kommen verschiedene Vorgehensweisen in Betracht:

So kann ein Poisson-ähnliches Modell verwendet werden, das einen zusätzlichen, aus den Daten zu schätzenden Streuungsparameter ϕ besitzt, mit dem für die bedingte Varianz $\sigma^2 = \phi\mu$ gilt. Hierfür ist das Argument `family` von `glm()` auf `quasipoisson(link="log")` zu setzen.²³ Dies führt zu identischen Parameterschätzungen b , jedoch zu anderen geschätzten Streuungen $\hat{\sigma}_b$ und damit zu anderen Ergebnissen der inferenzstatistischen Tests (vgl. Abschn. 8.1.6 für weitere Hinweise zu quasi-Familien).

```
> glmFitQP <- glm(Y ~ X1 + X2, family=quasipoisson(link="log"),
+                   data=dfCount)

> summary(glmFitQP)
Call:
glm(formula = Y ~ X1 + X2, family = quasipoisson(link = "log"),
     data = dfCount)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-3.1695	-1.4608	-0.2707	0.7173	3.4757

Coefficients:

Estimate	Std. Error	t value	Pr(> t)
----------	------------	---------	----------

²³Bei der Verwendung von `vglm()` aus dem Paket `VGAM` ist das Argument `family` auf `quasipoissonff` zu setzen.

```
(Intercept) 0.14455   0.12943   1.117    0.265
X1          -0.26240  0.02742  -9.569   <2e-16 ***
X2          0.02037   0.01405   1.450    0.149

(Dispersion parameter for quasipoisson family taken to be 1.697012)

Null deviance: 536.93 on 199 degrees of freedom
Residual deviance: 374.78 on 197 degrees of freedom
AIC: NA
```

Für eine adjustierte Poisson-Regression kann auch ein separater robuster Streuungsschätzer verwendet werden, wie ihn etwa das Paket `sandwich` mit `vcovHC()` bereitstellt. Wald-Tests der Parameter lassen sich dann mit `coeftest(glm-Modell, vcov=(Schätzer))` aus dem Paket `lmtest` durchführen, wobei für das Argument `vcov` das Ergebnis von `vcovHC()` anzugeben ist.

```
> library(sandwich)                                # für vcovHC()
> hcSE <- vcovHC(glmFitP, type="HC0")           # robuste SE-Schätzung
> library(lmtest)                                 # für coeftest()
> coeftest(glmFitP, vcov=hcSE)                  # Parametertests
z test of coefficients:
```

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	0.144554	0.135709	1.0652	0.2868
X1	-0.262399	0.028753	-9.1259	<2e-16 ***
X2	0.020368	0.013097	1.5552	0.1199

Eine Alternative ist die Regression mit der Annahme, dass die bedingte Verteilung von Y eine negative Binomialverteilung ist, die einen eigenen Dispersionsparameter θ besitzt. Sie verallgemeinert die Poisson-Verteilung mit demselben Erwartungswert μ , ihre Varianz ist jedoch mit $\mu + \frac{\mu^2}{\theta} = \mu(1 + \frac{\mu}{\theta})$ um den Faktor $1 + \frac{\mu}{\theta}$ größer. Die negative Binomial-Regression lässt sich mit `glm.nb()` aus dem Paket `MASS` anpassen.²⁴

```
> library(MASS)                                    # für glm.nb()
> glmFitNB <- glm.nb(Y ~ X1 + X2, data=dfCount)
> summary(glmFitNB)
Call:
glm.nb(formula = Y ~ X1 + X2, data = dfCount, init.theta = 3.852967973,
       link = log)

Deviance Residuals:
      Min        1Q     Median        3Q       Max
-2.6146  -1.2540  -0.2149   0.5074   2.7341

Coefficients:
```

	Estimate	Std. Error	z value	Pr(> z)
--	----------	------------	---------	----------

²⁴Bei der Verwendung von `vglm()` aus dem Paket `VGAM` ist das Argument `family` auf `negbinomial` zu setzen.

```
(Intercept) 0.003337 0.130007 0.026 0.9795
X1          -0.302121 0.029972 -10.080 <2e-16 ***
X2          0.025419 0.014649 1.735 0.0827 .

(Dispersion parameter for Negative Binomial(3.853) family taken to be 1)

Null deviance: 353.34 on 199 degrees of freedom
Residual deviance: 250.31 on 197 degrees of freedom
AIC: 801.11

Number of Fisher Scoring iterations: 1

Theta:  3.85
Std. Err.: 1.12
2 x log-likelihood: -793.113
```

Das Ergebnis nennt neben den bekannten Kennwerten unter `Theta` eine Schätzung für den Dispersionsparameter θ . Das von `glm.nb()` erzeugte Objekt kann an die Funktion `odTest()` aus dem Paket `pscl` (Zeileis, Kleiber & Jackman, 2008) übergeben werden. Sie führt dann einen Test auf Overdispersion durch, der auf dem Vergleich der Anpassungsgüte von Poisson- und negativem Binomial-Modell beruht.

```
> library(pscl)                                     # für odTest()
> odTest(glmFitNB)                                # overdispersion-Test
Likelihood ratio test of H0: Poisson, as restricted NB model:
n.b., the distribution of the test-statistic under H0 is non-standard
e.g., see help(odTest) for details/references

Critical value of test statistic at the alpha= 0.05 level: 2.7055
Chi-Square Test Statistic = 26.1033 p-value = 1.618e-07
```

8.4.4 Zero-inflated Poisson-Regression

Eine mögliche Quelle für starke Streuungen von Zähldaten sind gehäuft auftretende Nullen. Sie können das Ergebnis eines zweistufigen Prozesses sein, mit dem die Daten zustande kommen. Wird etwa erfasst, wie häufig Personen im Beruf befördert werden oder ihren Arbeitsplatz wechseln, ist zunächst relevant, ob sie überhaupt jemals eine Arbeitsstelle erhalten haben. Diese Eigenschaft wird vermutlich durch qualitativ andere Prozesse bestimmt als jene, die bei tatsächlich Erwerbstätigen die Anzahl der unterschiedlichen Positionen beeinflussen. Die Daten können damit als Ergebnis einer Mischung von zwei Verteilungen verstanden werden: Zum einen kommen Nullen durch Personen zustande, die ein notwendiges Kriterium für einen auch zu echt positiven Häufigkeiten führenden Prozess nicht erfüllen. Zum anderen verursachen Personen, die ein solches Kriterium erfüllen, ihrerseits Nullen sowie zusätzlich echt positive Häufigkeiten. Für die Daten dieser zweiten Gruppe von Personen kann nun wieder angenommen werden, dass sie sich durch eine Poisson- oder negative Binomialverteilung beschreiben lassen.

In der geschilderten Situation kommen *zero-inflated* Modelle in Betracht. Sie sorgen durch die Annahme einer Mischverteilung letztlich dafür, dass im Modell eine deutlich höhere Auftretenswahrscheinlichkeit von Nullen möglich ist, als es zur Verteilung der echt positiven Häufigkeiten passt.

Die zero-inflated Poisson-Regression eignet sich für Situationen, in denen keine starke overdispersion zu vermuten ist und kann mit `zeroinfl()` aus dem Paket `pscl` angepasst werden (für Details vgl. `vignette("countreg")`).²⁵

```
> zeroinfl(<Modellformel>, dist = <Verteilung>, offset = <offset>,
+           data = <Datensatz>)
```

Die Modellformel besitzt hier die Form $\langle AV \rangle \sim \langle UV \rangle + \langle 0\text{-Anteil} \rangle$. Für $\langle 0\text{-Anteil} \rangle$ ist ein Prädiktor zu nennen, der im Modell separat den Anteil der „festen“ Nullen an den Daten kontrolliert, die von jenen Personen stammen, die prinzipiell keine echt positiven Daten liefern können. Im einfachsten Fall ist dies der absolute Term 1, der zu einem Binomialmodell passt, das für alle Beobachtungen dieselbe Wahrscheinlichkeit vorsieht, zu dieser Gruppe zu gehören. In komplizierteren Situationen könnte die Gruppenzugehörigkeit analog zu einer separaten logistischen Regression durch einen eigenen Prädiktor vorhergesagt werden. Das Argument `dist` ist auf "poisson" zu setzen, `data` erwartet einen Datensatz mit den Variablen aus der Modellformel. Wie bei `glm()` existiert ein Argument `offset` für die Analyse von Ereignisraten (vgl. Abschn. 8.4.2).

```
> library(pscl)                                     # für zeroinfl()
> ziFitP <- zeroinfl(Y ~ X1 + X2 + 1, dist = "poisson", data = dfCount)
> summary(ziFitP)
Call:
zeroinfl(formula = Y ~ X1 + X2 + 1, data = dfCount, dist = "poisson")

Pearson residuals:
    Min      1Q  Median      3Q     Max 
-1.4805 -0.9447 -0.1574  0.6397  3.8149 

Count model coefficients (poisson with log link):
            Estimate Std. Error z value Pr(>|z|)    
(Intercept) 0.50809   0.13063   3.890  0.0001 *** 
X1          -0.20443   0.02658  -7.692 1.45e-14 *** 
X2          0.01781   0.01156   1.541   0.1233  
                                                        
Zero-inflation model coefficients (binomial with logit link):
            Estimate Std. Error z value Pr(>|z|)    
(Intercept) -1.4957    0.2627  -5.693 1.25e-08 *** 
                                                        
Number of iterations in BFGS optimization: 10
Log-likelihood: -393.1 on 4 Df
```

²⁵Bei der Verwendung von `vglm()` aus dem Paket `VGAM` ist das Argument `family` auf `zipoissonff` zu setzen.

Separate Wald-Tests der Parameter lassen sich mit `coeftest()` aus dem Paket `lmtest` durchführen, Likelihood-Quotienten-Tests für nested Modelle können mit `lrtest()` aus demselben Paket vorgenommen werden.

Die zero-inflated negative Binomial-Regression kann in Situationen mit gehäuft auftretenden Nullen und einer deutlichen overdispersion zum Einsatz kommen. Auch für sie eignet sich `zeroinfl()`, wobei das Argument `dist="negbin"` zu setzen ist.²⁶

```
> ziFitNB <- zeroinfl(Y ~ X1 + X2 | 1, dist="negbin", data=dfCount)
> summary(ziFitNB)
```

Call:

```
zeroinfl(formula = Y ~ X1 + X2 | 1, data = dfCount, dist = "negbin")
```

Pearson residuals:

Min	1Q	Median	3Q	Max
-1.4548	-0.9007	-0.1515	0.6482	3.9289

Count model coefficients (negbin with log link):

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	0.36993	0.17740	2.085	0.0370 *
X1	-0.23375	0.03738	-6.254	4e-10 ***
X2	0.02111	0.01320	1.599	0.1098
Log(theta)	2.65338	0.81911	3.239	0.0012 **

Zero-inflation model coefficients (binomial with logit link):

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	-1.7486	0.3924	-4.457	8.33e-06 ***

Theta = 14.2019

Number of iterations in BFGS optimization: 20

Log-likelihood: -392 on 5 Df

Der Vuong-Test vergleicht die Anpassungsgüte eines von `glm()` erstellten Poisson-Modells mit jener des durch `zeroinfl()` erstellten zero-inflated Poisson-Modells und kann mit `vuong()` aus dem Paket `pscl` durchgeführt werden. Analog testet `vuong()` auch ein mit `glm.nb()` angepasstes negatives Binomial-Modell gegen das zero-inflated negative Binomial-Modell aus `zeroinfl()`.

```
# Poisson-Modell vs. zero-inflated Poisson-Modell
> library(pscl)                                     # für vuong()
> vuong(ziFitP, glmFitP)
Vuong Non-Nested Hypothesis Test-Statistic: -0.4582521
(test-statistic is asymptotically distributed N(0,1) under the
null that the models are indistinguishable)
in this case:
model2 > model1, with p-value 0.32339
```

²⁶Bei der Verwendung von `vglm()` aus dem Paket `VGAM` ist das Argument `family` auf `zinegbinomial` zu setzen.

```
# negatives Binomial- vs. zero-inflated negatives Binomial-Modell
> vuong(ziFitNB, glmFitNB)
Vuong Non-Nested Hypothesis Test-Statistic: -1.382096
(test-statistic is asymptotically distributed N(0,1) under the
null that the models are indistinguishable)
in this case:
model2 > model1, with p-value 0.083471
```

Eine Alternative zu zero-inflated Modellen bei gehäuft auftretenden Nullen ist die Hurdle-Regression, die von `hurdle()` aus dem Paket `pscl` umgesetzt wird.

8.4.5 Zero-truncated Poisson-Regression

Im Gegensatz zum Umstand gehäuft auftretender Nullen erzwingt der Aufbau mancher Untersuchungen, dass *keine* Nullen beobachtet werden können. Dies ist etwa der Fall, wenn mindestens ein Ereignis vorliegen muss, damit eine Untersuchungseinheit in der Erhebung berücksichtigt wird: So ließe sich an bereits stationär aufgenommenen Patienten die Anzahl der Tage erheben, die sie im Krankenhaus verbringen, oder es könnte die Anzahl geborener Hundewelpen eines Wurfes erfasst werden. Zur Modellierung solcher Variablen kommen beschränkte (*zero-truncated*) Verteilungen in Frage, die den Wert 0 nur mit Wahrscheinlichkeit Null annehmen. Dazu zählen die zero-truncated Poisson-Verteilung – für Situationen ohne overdispersion – sowie die zero-truncated negative Binomialverteilung – bei overdispersion. Um sie bei der Modellanpassung mit `vglm()` aus dem Paket `VGAM` zu verwenden, ist das Argument `family` auf `pospoisson` bzw. auf `posnegbinomial` zu setzen.

8.5 Log-lineare Modelle

Log-lineare Modelle analysieren den Zusammenhang mehrerer kategorialer Variablen auf Basis ihrer gemeinsamen Häufigkeitsverteilung (Agresti, 2007). Sie sind für Daten geeignet, die sich als mehrdimensionale Kontingenztafeln absoluter Häufigkeiten darstellen lassen und verallgemeinern damit klassische nonparametrische Tests auf Unabhängigkeit bzw. auf Gleichheit von bedingten Verteilungen (vgl. Abschn. 10.2, 10.3.2).

8.5.1 Modell

Aus Sicht der Datenerhebung können Kontingenztafeln absoluter Häufigkeiten aus drei Situationen entstehen. Sie sollen hier an einer $(P \times Q)$ -Kontingenztafel der kategorialen Variablen

X_1 und X_2 erläutert werden.

	$x_{2,1}$...	$x_{2,q}$...	$x_{2,Q}$	Summe
$x_{1,1}$	f_{11}	...	f_{1k}	...	f_{1q}	f_{1+}
\vdots	\vdots	..	\vdots	..	\vdots	\vdots
$x_{1,p}$	f_{j1}	...	f_{jk}	...	f_{jq}	f_{p+}
\vdots	\vdots	..	\vdots	..	\vdots	\vdots
$x_{1,P}$	f_{p1}	...	f_{pk}	...	f_{pq}	f_{P+}
Summe	f_{+1}	...	f_{+q}	...	f_{+Q}	N

- Im *produkt-multinomialen* Erhebungsschema ist X_1 ein fester Faktor mit vorgegebenen Gruppenhäufigkeiten f_{p+} , aus deren Summe sich die feste Anzahl N von Beobachtungsobjekten ergibt – etwa in einem Experiment mit kontrollierter Zuweisung zu Versuchsbedingungen. Erhoben wird die kategoriale Variable X_2 , deren Verteilung über die Gruppen j i. S. der Auftretenshäufigkeiten ihrer Kategorien f_{+q} zufällig ist.
- Im *multinomialen* Erhebungsschema werden mehrere kategoriale Variablen X_j gleichzeitig an einer festen Anzahl N von Beobachtungsobjekten erhoben, etwa im Rahmen einer Befragung. Die Kontingenztafel entsteht aus der Kreuzklassifikation der X_j und gibt die Häufigkeiten an, mit der Kombinationen ihrer Kategorien auftreten. Dabei sind die f_{p+} und f_{+q} zufällig. In diesem Schema werden keine Einflussbeziehungen der X_j auf eine separate Variable untersucht, sondern Zusammenhänge zwischen den X_j selbst.
- Im *Poisson*-Erhebungsschema wird gleichzeitig mit den Ausprägungen kategorialer Einflussgrößen X_j eine separate Variable Y als Anzahl bestimmter Ereignisse erhoben. Die Kontingenztafel stellt die Anzahl der Ereignisse in den Gruppen dar, die durch die Kombination der Stufen der X_j entstehen. Dabei ist die Gesamthäufigkeit der Ereignisse N wie auch ihre Verteilung in den Gruppen f_{p+} und f_{+q} zufällig. In diesem Schema nimmt man eine Poisson-Verteilung für N an, womit jedes Ereignis unabhängig voneinander eintritt und als einzelne Beobachtung der Merkmalskombinationen der X_j zählt. Bedingt auf N liegt dann wieder ein multinomiales Schema vor.

Das log-lineare Modell ist ein lineares Modell für $\ln \mu_{jk}$, den logarithmierten Erwartungswert einer Zellhäufigkeit f_{jk} . Formuliert als Regression mit Treatment-Kontrasten hat es dieselbe Form wie das der Poisson-Regression (vgl. Abschn. 8.4.1). Es ist jedoch üblicher, es wie die mehrfaktorielle Varianzanalyse (vgl. Abschn. 7.5) mit Effektcodierung (vgl. Abschn. 12.9.2) zu parametrisieren. Mit der Stichprobengröße N , der Zellwahrscheinlichkeit p_{jk} und den Randwahrscheinlichkeiten p_{p+}, p_{+q} analog zu den Häufigkeiten gilt dafür zunächst:

$$\mu_{jk} = N p_{jk} = N p_{p+} p_{+q} \frac{p_{jk}}{p_{p+} p_{+q}}$$

$$\ln \mu_{jk} = \ln N + \ln p_{jk} = \ln N + \ln p_{p+} + \ln p_{+q} + (\ln p_{jk} - (\ln p_{p+} + \ln p_{+q}))$$

Das Modell für $\ln \mu_{jk}$ hat nun für eine zweidimensionale Kontingenztafel dieselbe Form wie jenes der zweifaktoriellen Varianzanalyse mit Effektcodierung. Dabei ist $\ln N$ analog zu μ , $\ln p_{p+}$

analog zu α_j (Zeileneffekt von Gruppe j des Faktors X_1), $\ln p_{+q}$ analog zu β_k (Spalteneffekt von Stufe k des Faktors X_2) und $\ln p_{jk} - (\ln p_{p+} + \ln p_{+q})$ analog zu $(\alpha\beta)_{jk}$ (Interaktionseffekt).²⁷

$$\begin{aligned}\mu_{jk} &= e^\mu e^{\alpha_j} e^{\beta_k} e^{(\alpha\beta)_{jk}} \\ \ln \mu_{jk} &= \mu + \alpha_j + \beta_k + (\alpha\beta)_{jk}\end{aligned}$$

Als Abweichung der logarithmierten Zellwahrscheinlichkeit von Additivität der logarithmierten Randwahrscheinlichkeiten drückt der Interaktionseffekt $(\alpha\beta)_{jk}$ den Zusammenhang von X_1 und X_2 aus, da bei Unabhängigkeit $p_{jk} = p_{p+} p_{+q}$ gilt – logarithmiert also $\ln p_{jk} = \ln p_{p+} + \ln p_{+q}$. In zweidimensionalen Kreuztabellen sind alle beobachtbaren Zellhäufigkeiten mit dem vollständigen Modell (Zeilen-, Spalten- und Interaktionseffekt) verträglich, das deswegen als *saturiert* bezeichnet wird und sich nicht testen lässt. Dagegen sind eingeschränkte Modelle wie das der Unabhängigkeit statistisch prüfbar. Für höherdimensionale Kontingenztafeln gilt dies analog, wobei kompliziertere Abhängigkeitsbeziehungen durch die Interaktionen erster und höherer Ordnung ausgedrückt werden können.

Für den Spezialfall zweidimensionaler Kreuztabellen ist das beschriebene Unabhängigkeitsmodell im multinomialen Erhebungsschema dasselbe, das vom χ^2 -Test auf Unabhängigkeit geprüft wird (vgl. Abschn. 10.2.1). Im produkt-multinomialen Erhebungsschema ist es analog dasselbe, das der χ^2 -Test auf Gleichheit von bedingten Verteilungen testet (vgl. Abschn. 10.2.2).

8.5.2 Modellanpassung

Log-lineare Modelle lassen sich mit der Funktion `loglm()` aus dem Paket **MASS** testen, in der die Modelle analog zu `lm()` formuliert werden können.²⁸

```
> loglm(<Modellformel>, data=<Kreuztabelle>)
```

Stammen die für `data` zu übergebenden Daten aus einer Kreuztabelle absoluter Häufigkeiten, kann das erste Argument eine Modellformel ohne Variable links der `~` sein. Besitzt die Kreuztabelle Namen für Zeilen und Spalten, sind diese mögliche Vorhersageterme in der Modellformel. Alternativ stehen die Ziffern 1, 2, ... für die Effekte der Zeilen, Spalten, etc. `data` kann auch ein Datensatz sein, der Spalten für die kategorialen Variablen X_j sowie für die Auftretenshäufigkeit Y jeder ihrer Stufenkombination besitzt (vgl. Abschn. 2.10.5). Wie gewohnt bildet Y dann die linke Seite der Modellformel und eine Kombination der X_j die rechte Seite.

Als Beispiel soll die $(2 \times 2 \times 6)$ -Kreuztabelle **UCBAdmissions** dienen, die im Basisumfang von R enthalten ist. Aufgeschlüsselt nach Geschlecht (Variable `Gender`) vermerkt sie die Häufigkeit, mit der Bewerber für die sechs größten Fakultäten (Variable `Dept`) an der University of California Berkeley 1973 angenommen bzw. abgelehnt wurden (Variable `Admit`).

²⁷ Anders als in der Varianzanalyse gibt es jedoch im log-linearen Modell nur eine Beobachtung pro Zelle, die Rolle der abhängigen Variable der Varianzanalyse hat im log-linearen Modell die logarithmierte Auftretenshäufigkeit der zur Zelle gehörenden Kombination von Faktorstufen.

²⁸ `loglm()` basiert auf `loglin()`, bietet jedoch die von Regression und Varianzanalyse vertraute Methode, eine Modellformel zur Beschreibung des log-linearen Modells zu verwenden.

```
> str(UCBAdmissions) # Struktur der Kreuztabelle
table [1:2, 1:2, 1:6] 512 313 89 19 353 207 17 8 120 205 ...
- attr(*, "dimnames")=List of 3
..$ Admit : chr [1:2] "Admitted" "Rejected"
..$ Gender: chr [1:2] "Male" "Female"
..$ Dept : chr [1:6] "A" "B" "C" "D" ...
# teste Modell der vollständigen Unabhängigkeit = Additivität
> (llFit <- loglm(~ Admit + Dept + Gender, data=UCBAdmissions))
Call:
loglm(formula = ~Admit + Dept + Gender, data = UCBAdmissions)

Statistics:
X^2 df P(> X^2)
Likelihood Ratio 2097.671 16 0
Pearson 2000.328 16 0

# wandle Kontingenztafel in Datensatz um
> UCBAdf <- as.data.frame(UCBAdmissions)
> loglm(Freq ~ Admit + Dept + Gender, data=UCBAdf) # ...
```

Die Ausgabe von `loglm()` nennt die Kennwerte des Likelihood-Quotienten-Tests sowie des Pearson χ^2 -Tests der Hypothese, dass das angegebene Modell stimmt. Der Wert der Teststatistik steht in der Spalte X^2 , die zugehörigen Freiheitsgrade unter `df` und der p -Wert unter `P(> X^2)`. Mit Hilfe von `coef(loglm-Modell)` erhält man zusätzlich Schätzungen für die durch die Modellformel spezifizierten Koeffizienten α_j, β_k, \dots . Dabei ist zu beachten, dass `loglm()` die Effektcodierung verwendet, die geschätzten Zeilen-, Spalten- und Schichten-Parameter summieren sich also pro Variable zu 0 (vgl. Abschn. 12.9.2).

```
> (llCoef <- coef(llFit))
$`(Intercept)`
[1] 5.177567

$Admit
  Admitted   Rejected
-0.2283697  0.2283697

$Gender
  Male    Female
0.1914342 -0.1914342

$Dept
      A          B          C          D          E          F
0.23047857 -0.23631478  0.21427076  0.06663476 -0.23802565 -0.03704367
```

Die gemeinsamen Häufigkeiten mehrerer kategorialer Variablen lassen sich mit `mosaicplot()` in einem Mosaik-Diagramm darstellen, einer Erweiterung des `splineplot` für die gemeinsame

Verteilung zweier kategorialer Variablen (vgl. Abschn. 14.4.2). Zusammen mit dem Argument `shade=TRUE` sind die Argumente dieselben wie für `loglm()`, die zellenweisen Pearson-Residuen bzgl. des durch die Modellformel definierten Models werden dann farbcodiert dargestellt (Abb. 8.5).

```
# Mosaik-Diagramm der Häufigkeiten und Pearson-Residuen
> mosaicplot(~ Admit + Dept + Gender, shade=TRUE, data=UCBAdmissions)
```

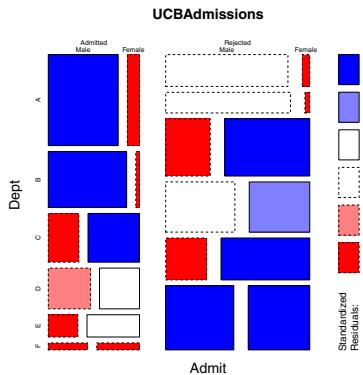


Abbildung 8.5: Mosaik-Diagramm der gemeinsamen Häufigkeiten der Kontingenztafel UCBAdmissions inkl. Farbcodierung der Pearson-Residuen vom Modell der Unabhängigkeit.

Wie in der Poisson-Regression erhält man mit `residuals(loglm-Modell, type="<Typ>")` die Residuen, hier allerdings nicht pro Beobachtung, sondern pro Zelle der Kontingenztafel. Mit `type="pearson"` sind dies die Pearson-Residuen, deren Quadratsumme gleich der Pearson-Teststatistik in der Ausgabe von `loglm()` ist. Für zweidimensionale Kontingenztafeln ist diese Quadratsumme gleich der Teststatistik des χ^2 -Tests auf Unabhängigkeit mit `chisq.test()` (vgl. Abschn. 10.2.1). Analog erhält man mit `type="deviance"` die Devianz-Residuen, deren Quadratsumme gleich der Likelihood-Ratio-Teststatistik in der Ausgabe von `loglm()` ist.

```
> sum(residuals(llFit, type="deviance"))^2
[1] 2097.671
```

```
> sum(residuals(llFit, type="pearson"))^2
[1] 2000.328
```

Anders als in der Poisson-Regression liefert ein mit `loglm()` angepasstes Modell keine Standardfehler der Parameterschätzungen. Um diese zu beurteilen, lässt sich das Modell jedoch als Poisson-Regression mit `glm()` formulieren. Da `glm()` in der Voreinstellung Treatment-Kontraste verwendet, sind die Parameterschätzungen zunächst andere, mit Effektcodierung jedoch dieselben. Vorhersage und Residuen stimmen mit der des log-linearen Modells überein.²⁹

²⁹Damit ist auch die Teststatistik des Likelihood-Quotienten-Tests im Prinzip identisch, da `loglm()` und `glm()` jedoch andere numerische Optimierungsverfahren zur Maximum-Likelihood-Schätzung verwenden, sind kleine Abweichungen möglich.

```
# Poisson-Regression mit Treatment-Kontrasten
> glmFitT <- glm(Freq ~ Admit+Dept+Gender, family=poisson(link="log"),
+                   data=UCBAdf)

# prüfe Gleichheit der Vorhersage mit log-linearem Modell
> all.equal(c(fitted(llFit)), fitted(glmFitT), check.attributes=FALSE)
[1] TRUE

> (glmTcoef <- coef(glmFitT))                      # Parameterschätzungen
Coefficients:
(Intercept)  AdmitRejected      DeptB      Dept      DeptD      DeptE
  5.37111     0.45674   -0.46679   -0.01621   -0.16384   -0.46850

  DeptF  GenderFemale
-0.26752     -0.38287
```

Bei Treatment-Kontrasten ist der absolute Term gleich der Vorhersage in der Zelle, die sich als Kombination aller Referenzkategorien der beteiligten Faktoren (in der Voreinstellung jeweils die erste Faktorstufe) ergibt. Die geschätzten Koeffizienten geben jeweils die geschätzte Abweichung für eine Faktorstufe von der Referenzkategorie an.

```
# Umrechnung mu_jk Effektcodierung aus loglm() in Treatment-Kontraste
# Zelle aus Kombination der Referenz-Kategorien
> glmTcoef["(Intercept)"]                                # Treatment-Kontraste
(Intercept)
  5.37111

> llCoef`~(Intercept)` + llCoef$Admit["Admitted"] +    # Effektcodierung
+     llCoef$Gender["Male"] + llCoef$Dept["A"]
Admitted
  5.37111

# mu_jk für Admit = "Admit", Gender = "Female", Dept = "C"
# Treatment-Kontraste
> glmTcoef["(Intercept)"] + glmTcoef["DeptC"] + glmTcoef["GenderFemale"]
(Intercept)
  4.972034

> llCoef`~(Intercept)` + llCoef$Admit["Admitted"] +    # Effektcodierung
+     llCoef$Dept["C"] + llCoef$Gender["Female"]
Admitted
  4.972034
```

Auch mit `glm()` lassen sich die Parameter mit Effektcodierung schätzen (vgl. Abschn. 7.5.2). Dabei fehlt die Parameterschätzung für die jeweils letzte Faktorstufe in der Ausgabe, da sie sich aus der Nebenbedingung ergibt, dass sich die Parameter pro Faktor zu Null summieren. Die Standardfehler der Parameterschätzungen sowie den zugehörigen Wald-Test extrahiert man

mit `coef(summary(glm-Modell))`). Die Einträge für den absoluten Term (`Intercept`) sind hier ohne Bedeutung, da $\ln N$ im log-linearen Modell fest vorgegeben ist und nicht geschätzt werden muss.

```
# Poisson-Regression mit Effektcodierung
> glmFitE <- glm(Freq ~ Admit + Dept + Gender, family=poisson(link="log"),
+                   contrasts=list(Admit=contr.sum,
+                                     Dept=contr.sum,
+                                     Gender=contr.sum), data=UCBAdf)

# Parameterschätzungen mit Standardfehlern und Wald-Tests
> coef(summary(glmFitE))
            Estimate Std. Error     z value   Pr(>|z|)
(Intercept) 5.17756677 0.01577888 328.132716 0.000000e+00
Admit1      -0.22836970 0.01525343 -14.971696 1.124171e-50
Dept1        0.23047857 0.03071783   7.503086 6.233240e-14
Dept2        -0.23631478 0.03699431  -6.387869 1.682136e-10
Dept3        0.21427076 0.03090740   6.932668 4.129780e-12
Dept4        0.06663476 0.03272311   2.036321 4.171810e-02
Dept5        -0.23802565 0.03702165  -6.429363 1.281395e-10
Gender1      0.19143420 0.01513732  12.646505 1.169626e-36
```

Kapitel 9

Survival-Analyse

Die Survival-Analyse modelliert Überlebenszeiten (Hosmer Jr, Lemeshow & May, 2008; Klein & Moeschberger, 2003). Diese geben allgemein an, wieviel Zeit bis zum Eintreten eines bestimmten Ereignisses verstrichen ist und sollen hier deshalb gleichbedeutend mit *Ereigniszeiten* sein. Es kann sich dabei etwa um die Zeitspanne handeln, die ein Patient nach einer Behandlung weiter am Leben ist, um die verstrichene Zeit, bis ein bestimmtes Bauteil im Gebrauch einen Defekt aufweist, oder um die Dauer, die ein Kleinkind benötigt, um ein vordefiniertes Entwicklungsziel zu erreichen – z. B. einen Mindestwortschatz besitzt. Bei der Analyse von Überlebenszeiten kann sowohl die Form ihres grundsätzlichen Verlaufs von Interesse sein, als auch inwiefern ihr Verlauf systematisch von Einflussgrößen abhängt.

Die folgenden Auswertungen verwenden Funktionen des Pakets `survival`, das im Basisumfang von R enthalten ist. Seine Anwendung wird vertiefend in Harrell Jr (2015) behandelt.

9.1 Verteilung von Ereigniszeiten

Überlebenszeiten lassen sich äquivalent durch verschiedene Funktionen beschreiben, die jeweils andere Eigenschaften ihrer Verteilung hervortreten lassen: Die Überlebenszeit selbst sei T mit Werten ≥ 0 und einer – hier als stetig vorausgesetzten – Dichtefunktion $f(t)$.¹ Die zugehörige Verteilungsfunktion $F(t) = P(T \leq t)$ liefert die Wahrscheinlichkeit, mit der T höchstens den Wert t erreicht. Die monoton fallende Survival-Funktion $S(t) = P(T > t) = 1 - F(t)$ gibt an, mit welcher Wahrscheinlichkeit die Überlebenszeit größer als t ist. Dabei wird $S(0) = 1$ vorausgesetzt, zum Zeitpunkt $t = 0$ soll das Ereignis also noch nicht eingetreten sein. Schließlich drückt die Hazard-Funktion $\lambda(t)$ die unmittelbare Ereignisrate zum Zeitpunkt t aus.

$$\lambda(t) = \lim_{\Delta_t \rightarrow 0^+} \frac{P(t \leq T < t + \Delta_t | T \geq t)}{\Delta_t} = \frac{P(t \leq T < t + \Delta_t)/\Delta_t}{P(T > t)} = \frac{f(t)}{S(t)}, \quad t \geq 0$$

$\lambda(t)$ ist bei kleiner werdender Intervallbreite Δ_t der Grenzwert für die bedingte Wahrscheinlichkeit pro Zeiteinheit, dass das Ereignis unmittelbar eintritt, wenn es bis zum Zeitpunkt t noch nicht eingetreten ist. Bezeichnet $\#\text{Personen} | \text{Ereignis} \in [t, t + \Delta_t]$ die Anzahl der Personen, bei denen das Ereignis im Intervall $[t, t + \Delta_t]$ eintritt und $\#\text{Personen} | \text{Ereignis} \geq t$ die Anzahl der

¹ $f(t) = \lim_{\Delta_t \rightarrow 0^+} \frac{P(t \leq T < t + \Delta_t)}{\Delta_t}$ in Abhängigkeit von der Zeit t und der Intervallbreite Δ_t .

Personen, bei denen das Ereignis zum Zeitpunkt t noch eintreten kann, lässt sich das hazard auf empirischer Ebene wie folgt formulieren:²

$$\hat{\lambda}(t) = \frac{\#\text{Personen} \mid \text{Ereignis} \in [t, t + \Delta_t)}{\#\text{Personen} \mid \text{Ereignis} \geq t} \cdot \frac{1}{\Delta_t}$$

$\hat{\lambda}(t)$ gibt also an, bei welchem Anteil verbleibender Personen ohne Ereignis bis zum Zeitpunkt t das Ereignis pro Zeiteinheit eintritt. Bei Werten der monoton steigenden kumulativen Hazard-Funktion $\Lambda(t) = -\ln S(t)$ handelt es sich um das bis zum Zeitpunkt t kumulierte Risiko, dass sich das Ereignis unmittelbar ereignet. Umgekehrt gilt $S(t) = e^{-\Lambda(t)}$.

Hazard- und Survival-Funktion bedingen einander. Nimmt man eine über die Zeit konstante Ereignisrate $\lambda(t) = \frac{1}{b}$ an (wie etwa beim radioaktiven Zerfall), impliziert dies eine exponentiell verteilte Überlebenszeit T (vgl. Abschn. 9.5). Die bedingte Wahrscheinlichkeit, dass ein noch nicht eingetretenes Ereignis unmittelbar auf t folgt, wäre damit unabhängig von der bereits verstrichenen Zeit t . Mit der Annahme, dass die logarithmierte Ereignisrate linear von t abhängt ($\ln \lambda(t) = \alpha + \rho t$ mit y -Achsenabschnitt α und Steigung ρ), ergibt sich entsprechend eine Gompertz-Verteilung von T . Analog führt die Annahme, dass die logarithmierte Ereignisrate linear mit der logarithmierten Zeit zusammenhängt ($\ln \lambda(t) = \alpha + \rho \ln t$), zu einer Weibull-Verteilung von T (vgl. Abschn. 9.5). Bei einem positivem ρ würde in beiden Fällen das hazard mit der Zeit ansteigen, was oft in Situationen angemessen ist, in denen das Eintreten des Ereignisses mit kontinuierlichen Reifungs- oder Abnutzungsprozessen zusammenhängt.

9.2 Zensierte und gestutzte Ereigniszeiten

Um festzustellen, wann ein Zielereignis eintritt, werden die untersuchten Objekte über eine gewisse Zeit hinweg beobachtet – etwa wenn bei aus einer stationären Behandlung entlassenen Patienten mit Substanzmissbrauch erhoben wird, ob sich innerhalb eines Zeitraums ein Rückfall ereignet. Meist weisen empirisch erhobene Überlebenszeiten dabei die Besonderheit auf, dass von einigen Beobachtungseinheiten die Zeit bis zum Eintreten des Ereignisses unbekannt bleibt, was spezialisierte statistische Modelle notwendig macht.

Der Erhebungszeitraum ist oft begrenzt, so dass nicht für alle Untersuchungseinheiten das Ereignis auch tatsächlich innerhalb des Beobachtungszeitraums auftritt. Für solche *rechts-zensierte* Daten ist also nur bekannt, dass die Überlebenszeit den letzten Beobachtungszeitpunkt überschreitet, nicht aber ihr exakter Wert. Eine andere Ursache für rechts-zensierte Daten kann ein frühzeitiger dropout aus der Studie nach einem Umzug oder bei schwindender Motivation zur Teilnahme sein. *Links-zensierte* Daten entstehen, wenn das Ereignis bekanntermaßen bereits an einem unbekannten Zeitpunkt vor Beginn der Erhebung eingetreten ist. Daten werden als *links-gestutzt* bezeichnet, wenn sich das Ereignis bei manchen potentiellen Beobachtungseinheiten bereits vor Erhebungsbeginn ereignet und sie deswegen nicht mehr in der Studie berücksichtigt werden können. Während die Häufigkeit zensierter Beobachtungen in der Stichprobe bekannt ist, fehlt über gestutzte Daten jede Information.

² $\#\text{Personen} \mid \text{Ereignis} \geq t$ ist die Größe des *risk set* bzw. die Anzahl der Beobachtungsobjekte *at risk* zum Zeitpunkt t .

Wichtig für die Survival-Analyse ist die Annahme, dass der zur Zensierung führende Mechanismus unabhängig von Einflussgrößen auf die Überlebenszeit ist, Beobachtungsobjekte mit zensierter Überlebenszeit also kein systematisch anderes hazard haben. Diese Bedingung wäre etwa dann erfüllt, wenn zensierte Daten dadurch entstehen, dass eine Studie zu einem vorher festgelegten Zeitpunkt endet, bis zu dem nicht bei allen Untersuchungseinheiten das Ereignis eingetreten ist. Bewirkt eine Ursache dagegen sowohl das nahe bevorstehende Eintreten des Ereignisses selbst als auch den Ausfall von Beobachtungsmöglichkeiten, wäre die Annahme nicht-informativer Zensierung verletzt. So könnte eine steigende zeitliche Beanspruchung im Beruf bei Patienten mit Substanzmissbrauch einerseits dazu führen, dass die Bereitschaft zur Teilnahme an Kontrollterminen sinkt, andererseits könnte sie gleichzeitig die Wahrscheinlichkeit eines Rückfalls erhöhen. Wenn selektiv Beobachtungseinheiten mit erhöhtem hazard nicht mehr beobachtet werden können, besteht die Gefahr verzerrter Schätzungen des Verlaufs der Überlebenszeiten.

9.2.1 Zeitlich konstante Prädiktoren

Survival-Daten beinhalten Angaben zum Beobachtungszeitpunkt, zu den Prädiktoren der Überlebenszeit sowie eine Indikatorvariable dafür, ob das Ereignis zum angegebenen Zeitpunkt beobachtet wurde. Für die Verwendung in späteren Analysen sind Beobachtungszeitpunkt und Indikatorvariable zunächst in einem Survival-Objekt zusammenzuführen, das Informationen zur Art der Zensierung der Daten berücksichtigt. Dies geschieht für potentiell rechts-zensierte Daten mit `Surv()` aus dem Paket `survival` in der folgenden Form:

```
> Surv(<Zeitpunkt>, <Status>)
```

Als erstes Argument ist ein Vektor der Zeitpunkte $t_i > 0$ zu nennen, an denen das Ereignis bei den Objekten i eingetreten ist. Bei rechts-zensierten Beobachtungen ist dies der letzte bekannte Zeitpunkt, zu dem das Ereignis noch nicht eingetreten war. Die dabei implizit verwendete Zeitskala hat ihren Ursprung 0 beim Eintritt in die Untersuchung. Das zweite Argument ist eine numerische oder logische Indikatorvariable, die den Status zu den genannten Zeitpunkten angibt – ob das Ereignis also vorlag (1 bzw. `TRUE`) oder nicht (0 bzw. `FALSE` bei rechts-zensierten Beobachtungen).³

Für die folgende Simulation von Überlebenszeiten soll eine Weibull-Verteilung mit Annahme proportionaler hazards bzgl. der Einflussfaktoren (vgl. Abschn. 9.4) zugrunde gelegt werden (Abb. 9.1). Dafür sei der lineare Effekt der Einflussgrößen durch $\mathbf{X}\boldsymbol{\beta} = \beta_1 X_1 + \dots + \beta_j X_j + \dots + \beta_p X_p$ gegeben (ohne absoluten Term β_0 , vgl. Abschn. 12.9.1). Hier soll dafür ein kontinuierlicher Prädiktor sowie eine kategoriale UV mit 3 Stufen verwendet werden, wobei beide Variablen nicht über die Zeit variieren. Zusätzlich sei die Schichtung hinsichtlich des Geschlechts berücksichtigt.

```
> N    <- 180                                # Anzahl Personen
> P    <- 3                                   # Anzahl Stufen UV
> sex <- factor(sample(c("f", "m"), N, replace=TRUE))  # Geschlecht
> X    <- rnorm(N, 0, 1)                      # kont. Prädiktor
```

³Für Intervall-zensierte Daten vgl. `?Surv`. Vergleiche Abschn. 9.2.2 für zeitabhängige Prädiktoren und Fälle, in denen mehrere Ereignisse pro Beobachtungsobjekt möglich sind.

```
> IV <- factor(rep(LETTERS[1:P], each=N/P)) # UV Faktor

# Effekte der UV-Stufen: 1. Stufe = baseline -> Effekt 0
> IVeff <- c(0, -1, 1.5)

# zusammengefasster Effekt der Einflussgrößen mit zufälligem Fehler
> Xbeta <- 0.7*X + IVeff[unclass(IV)] + rnorm(N, 0, 2)
```

Weiter sei $U \sim \mathcal{U}(0, 1)$ eine gleichverteilte Zufallsvariable auf dem Intervall $[0, 1]$. Weibull-verteilte Überlebenszeiten können dann als Realisierung von $T = (-\ln(U)) b^a e^{-X^\beta \frac{1}{a}}$ simuliert werden (vgl. Abschn. 9.5). Die hier getroffene Wahl $a = 1.5$ führt dazu, dass $\ln \lambda(t)$ linear mit $\ln t$ ansteigt. Die Simulation von Überlebenszeiten mit anderen kumulativen Hazard-Funktionen $\Lambda(t)$ unter Annahme proportionaler hazards erfolgt allgemein mit $\Lambda^{-1}(-\ln(U) e^{-X^\beta})$ (Bender, Augustin & Blettner, 2005).

```
# Weibull-Verteilung zur Charakterisierung des baseline-hazards
> weibA <- 1.5 # Formparameter
> weibB <- 100 # Skalierungsparameter
> U <- runif(N, 0, 1) # gleichverteilte Var

# Überlebenszeiten - aufrunden für t > 0
> eventT <- ceiling((-log(U)*(weibB^weibA)*exp(-Xbeta))^(1/weibA))
> obsLen <- 120 # Beobachtungsdauer

# stelle kumulierte Verteilung der Überlebenszeiten dar
> plot(ecdf(eventT), xlim=c(0, 200), main="Kumulative
+ Überlebenszeit-Verteilung", xlab="t", ylab="F(t)")

> abline(v=obsLen, col="blue", lwd=2) # Untersuchungsende
> text(obsLen-5, 0.2, adj=1, labels="Ende Beobachtungszeit")
```

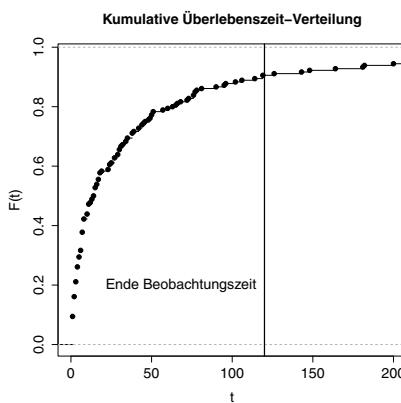


Abbildung 9.1: Kumulative Verteilung simulierter Survival-Daten mit Weibull-Verteilung

Der zur rechts-Zensierung führende Prozess soll hier ausschließlich das geplante Ende des Beobachtungszeitraums sein. Alle nach dem Endpunkt der Erhebung liegenden Überlebenszeiten werden daher auf diesen Endpunkt zensiert. Entsprechend wird das Ereignis nur beobachtet, wenn die Überlebenszeit nicht nach dem Endpunkt liegt.

```
> censT <- rep(obsLen, N)                                # Zensierungszeit
> obsT <- pmin(eventT, censT)                            # zensierte Ü-Zeit
> status <- eventT <= censT                             # Ereignis-Status
> dfSurv <- data.frame(obsT, status, sex, X, IV)        # Datensatz
> library(survival)                                       # für Surv()
> Surv(obsT, status)                                     # Survival-Objekt
[1] 63 25 73 120+ 4 39 4 1 10 11 2 120+ 7 29 95 ...
```

In der (hier gekürzten) Ausgabe des mit `Surv()` gebildeten Survival-Objekts sind zensierte Überlebenszeiten durch ein + kenntlich gemacht.

9.2.2 Daten in Zählprozess-Darstellung

Überlebenszeiten lassen sich samt der zugehörigen Werte für relevante Prädiktoren auch in der Zählprozess-Darstellung notieren. Für jedes Beobachtungsobjekt unterteilt diese Darstellung die Zeit in diskrete Intervalle (Start, Stop], für die jeweils angegeben wird, ob sich ein Ereignis im Intervall ereignet hat. Die Intervalle sind links offen und rechts geschlossen, zudem muss `Stop > Start` gelten, so dass keine Intervalle der Länge 0 auftauchen. Der Aufruf von `Surv()` erweitert sich dafür wie folgt:

```
> Surv(<Start>, <Stop>, <Status>)
```

`<Start>` gibt den Beginn eines Beobachtungsintervalls an. `<Stop>` ist wieder der beobachtete Ereignis-Zeitpunkt bzw. bei rechts-zensierten Daten der letzte bekannte Zeitpunkt, zu dem kein Ereignis vorlag. Intervalle für unterschiedliche Personen dürfen dabei unterschiedliche Grenzen besitzen. Anders als in der in Abschn. 9.2.1 vorgestellten Darstellung kann die hier implizit verwendete Zeitskala flexibel etwa das Alter eines Beobachtungsobjekts sein, die (vom Eintritt in die Untersuchung abweichende) Zeit seit einer Diagnose oder die absolute kalendarische Zeit. Ist das Ereignis im Intervall `(Start, Stop]` eingetreten, ist `<Status>` gleich 1 (bzw. `TRUE`), sonst 0 (bzw. `FALSE` bei rechts-zensierten Beobachtungen).

Durch die explizite Angabe des Beginns eines Beobachtungszeitraums gibt die Zählprozess-Darstellung Auskunft darüber, vor welchem Zeitpunkt keine Informationen darüber vorliegen, ob sich das Ereignis bereits einmal ereignet hat. Die Darstellungsform ist damit für links-gestützte Daten geeignet. Tabelle 9.1 zeigt die Darstellung am Beispiel von vier Beobachtungsobjekten aus zwei Gruppen, deren Lebensalter zu Beginn und zu Ende einer zenjährigen Untersuchung ebenso erfasst wurde wie der Ereignis-Status zu Ende der Untersuchung.

Die Zählprozess-Darstellung erlaubt es auch Erhebungssituationen abzubilden, in denen der Status von Beobachtungsobjekten an mehreren Zeitpunkten t_m ermittelt wird. So könnte etwa bei regelmäßigen Kontrollterminen nach einer Operation geprüft werden, ob eine bestimmte Krankheit wieder diagnostizierbar ist. Der Aufruf von `Surv()` hat dann die Form wie bei links-gestützten Beobachtungen, wobei pro Beobachtungsobjekt mehrere `(Start, Stop]` Intervalle

Tabelle 9.1: Zählprozess-Darstellung für vier links-gestützte und teilweise rechts-zensierte Beobachtungen mit Überlebenszeiten (Lebensalter) $\{47, 38^+, 41, 55\}$ aus zwei Gruppen in einer zenjährigen Untersuchung

Start	Stop	Gruppe	Status
37	47	Treatment	1
28	38	Treatment	0
31	41	Control	1
45	55	Control	1

vorliegen. Für den Untersuchungszeitpunkt t_m ist dabei $\langle \text{Start} \rangle$ der letzte zuvor liegende Untersuchungszeitpunkt t_{m-1} , während $\langle \text{Stop} \rangle$ t_m selbst ist. Für den ersten Untersuchungszeitpunkt t_1 ist t_0 der Eintritt in die Untersuchung, etwa das zugehörige kalendarische Datum oder das Alter eines Beobachtungsobjekts.

Wiederkehrende Ereignisse

Da der Ereignis-Status in der Zählprozess-Darstellung am Ende mehrerer Zeitintervalle erfasst werden kann, ist es auch möglich Ereignisse abzubilden, die sich pro Untersuchungseinheit potentiell mehrfach ereignen. Die spätere Analyse wiederkehrender Ereignisse muss dabei berücksichtigen, ob diese unabhängig voneinander eintreten, oder etwa in ihrer Reihenfolge festgelegt sind.

Bei mehreren Beobachtungszeitpunkten muss der Datensatz für die Zählprozess-Darstellung im Long-Format vorliegen, d. h. für jede Untersuchungseinheit mehrere Zeilen umfassen (vgl. Abschn. 3.3.9). Pro Beobachtungszeitpunkt t_m ist für jede Untersuchungseinheit eine Zeile mit den Werten t_{m-1} , t_m , den Werten der Prädiktoren sowie dem Ereignis-Status zu t_m in den Datensatz aufzunehmen. Zusätzlich sollte jede Zeile den Wert eines Faktors $\langle \text{ID} \rangle$ enthalten, der das Beobachtungsobjekt identifiziert (Tab. 9.2).

Tabelle 9.2: Zählprozess-Darstellung für zwei Beobachtungsobjekte aus zwei Gruppen am Ende von je drei zweijährigen Beobachtungsintervallen mit potentiell mehrfach auftretenden Ereignissen

ID	Start (t_{m-1})	Stop (t_m)	Gruppe	Status
1	37	39	Treatment	0
1	39	41	Treatment	1
1	41	43	Treatment	1
2	28	30	Control	0
2	30	32	Control	1
2	32	34	Control	0

Liegen die Daten im $\langle \text{Zeitpunkt} \rangle$, $\langle \text{Status} \rangle$ -Format vor, können sie mit `survSplit()` aus

dem Paket **survival** in Zählprozess-Darstellung mit mehreren Zeitintervallen pro Beobachtungsobjekt gebracht werden.

```
> survSplit(data=<Datensatz>, cut=<Grenzen>, end=<Zeitpunkt>,
+           event=<Status>, start=<Name>, id=<Name>,
+           zero=<Startzeit>)
```

Für **data** ist der Datensatz anzugeben. **cut** legt die linken Grenzen t_{m-1} der neu zu bildenden Intervalle fest, jedoch ohne die erste Grenze t_0 .⁴ Die Variable mit der (ggf. zensierten) Überlebenszeit aus **data** ist unter **end** zu nennen, die Variable mit dem zugehörigen Ereignis-Status unter **event**. **start** ist der Name der zu erstellenden Variable der linken Intervallgrenzen t_{m-1} . Soll eine Variable hinzugefügt werden, die jedem Intervall das zugehörige Beobachtungsobjekt zuordnet, muss deren Name für **id** genannt werden. Mit **zero** lässt sich in Form eines Vektors oder als – dann für alle Beobachtungen identische – Zahl angeben, was der Zeitpunkt t_0 des Beginns der Beobachtungen ist.

```
> library(survival)                                # für survSplit()
> dfSurvCP <- survSplit(dfSurv, cut=seq(30, 90, by=30), end="obsT",
+                        event="status", start="start", id="ID", zero=0)

# sortiere nach Beobachtungsobjekt und linken Intervallgrenzen
> idxOrd <- order(dfSurvCP$ID, dfSurvCP$start)
> head(dfSurvCP[idxOrd, ], n=7)
   obsT status sex      X  IV start   ID
1     30     0   f -1.3130607   A     0    1
181   60     0   f -1.3130607   A    30    1
361   63     1   f -1.3130607   A    60    1
10    11     1   f -1.2282824   A     0   10
100   27     1   m -0.1018403   B     0  100
101   30     0   m -0.4079027   B     0  101
281   42     1   m -0.4079027   B    30  101
```

Zeitabhängige Prädiktoren

In der Zählprozess-Darstellung ist es möglich, die Werte von zeitlich variablen Prädiktoren in die Daten aufzunehmen (Tab. 9.3). Eine spätere Analyse setzt dabei voraus, dass der Wert eines zeitabhängigen Prädiktors zum Zeitpunkt t_m nur Informationen widerspiegelt, die bis t_m vorlagen – aber nicht später. Eine zeitlich rückwirkende Kategorisierung von Untersuchungseinheiten in verschiedene Gruppen auf Basis ihres Verhaltens zu Studienende wäre etwa demnach unzulässig. Ebenso sind meist zeitabhängige Variablen problematisch, die weniger Einflussgröße bzw. Prädiktor, sondern eher Effekt oder Indikator eines Prozesses sind, der zu einem bevorstehenden Ereignis führt.

⁴Bei nicht wiederkehrenden Ereignissen ist die Einteilung der Gesamtbeobachtungszeit in einzelne, bündig aneinander anschließende Intervalle beliebig: Die einzelne Beobachtung im (**Zeitpunkt**), (**Status**)-Format (10, TRUE) ist sowohl äquivalent zu den zwei Beobachtungen in Zählprozess-Darstellung (0, 4, FALSE), (4, 10, TRUE) als auch zu den drei Beobachtungen (0, 2, FALSE), (2, 6, FALSE), (6, 10, TRUE).

Tabelle 9.3: Zählprozess-Darstellung für zeitabhängige Prädiktoren bei drei Beobachtungsobjekten mit Überlebenszeiten $\{3, 4^+, 2\}$

ID	Start (t_{m-1})	Stop (t_m)	Temperatur	Status
1	0	1	45	0
1	1	2	52	0
1	2	3	58	1
2	0	1	37	0
2	1	2	41	0
2	2	3	56	0
2	3	4	57	0
3	0	1	35	0
3	1	2	61	1

9.3 Kaplan-Meier-Analyse

9.3.1 Survival-Funktion und kumulatives hazard schätzen

Die Kaplan-Meier-Analyse liefert als nonparametrische Maximum-Likelihood-Schätzung der Survival-Funktion eine Stufenfunktion $\hat{S}(t)$, deren Stufen bei den empirisch beobachteten Überlebenszeiten liegen. Sie wird samt der punktweisen Konfidenzintervalle mit `survfit()` aus dem Paket `survival` berechnet.

```
> survfit(<Modellformel>, type="kaplan-meier", conf.type="<CI-Typ>")
```

Als erstes Argument ist eine Modellformel zu übergeben, deren linke Seite ein mit `Surv()` erstelltes Objekt ist (vgl. Abschn. 9.2). Die rechte Seite der Modellformel ist entweder der konstante Term 1 für eine globale Anpassung, oder besteht aus (zeitlich konstanten) Faktoren. In diesem Fall resultiert für jede Faktorstufe bzw. Kombination von Faktorstufen g eine separate Schätzung $\hat{S}_g(t)$. Das Argument `type` ist bei der Voreinstellung "kaplan-meier" zu belassen. Mit `conf.type` wird die Transformation für $S(t)$ angegeben, auf deren Basis die nach zugehöriger Rücktransformation gewonnenen Konfidenzintervalle für $S(t)$ konstruiert sind: Mit "plain" erhält man Intervalle auf Basis von $S(t)$ selbst. Geeigneter sind oft die durch "log" erzeugten Intervalle, die auf dem kumulativen hazard $\Lambda(t) = -\ln S(t)$ basieren. Mit "log-log" ergeben sich die Intervalle aus dem logarithmierten kumulativen hazard $\ln \Lambda(t) = \ln(-\ln S(t))$. Auf "none" gesetzt unterbleibt die Berechnung von Konfidenzintervallen.

```
# Schätzung von S(t) ohne Trennung nach Gruppen
> library(survival)                                     # für survfit()
> KM0 <- survfit(Surv(obsT, status) ~ 1, type="kaplan-meier",
+                  conf.type="log", data=dfSurv)

# Schätzung von S(t) getrennt nach Gruppen
> (KM <- survfit(Surv(obsT, status) ~ IV, type="kaplan-meier",
+                  conf.type="log", data=dfSurv))
```

```
Call: survfit(formula = Surv(obsT, status) ~ IV, data = dfSurv,
  type = "kaplan-meier", conf.type = "log")
```

	records	n.max	n.start	events	median	0.95LCL	0.95UCL
IV=A	60	60	60	55	13.5	8	31
IV=B	60	60	60	49	33.0	17	50
IV=C	60	60	60	59	5.5	4	11

Die Ausgabe gibt – ggf. getrennt für jede Gruppe – Auskunft über die Anzahl der Beobachtungen (**records**), die Anzahl der Ereignisse (**events**) und unter **median** eine Schätzung für den Median der Überlebenszeit (das 50%-Quantil der geschätzten Survival-Funktion $\hat{S}(t)$) gefolgt von den Grenzen des zugehörigen Konfidenzintervalls (**0.95LCL**, **0.95UCL**).

Analog zum geschätzten Median der Überlebenszeit ermittelt `quantile(<survfit-Objekt>, probs=(Quantile))` beliebige Quantile von $\hat{S}(t)$ – also Schätzungen für die Zeitpunkte, zu denen bei einem bestimmten Anteil der Beobachtungsobjekte ein Ereignis aufgetreten ist. In der Voreinstellung `conf.int=TRUE` erhält man zusätzlich die Grenzen des jeweiligen Konfidenzintervalls für ein Quantil. `summary(<survfit-Objekt>)` gibt detailliert Auskunft über die aufgetretenen Ereignis-Zeitpunkte t_i und Werte von $\hat{S}(t)$.

```
# Quantile der Überlebenszeit ohne Grenzen der Konfidenzintervalle
> quantile(KM0, probs=c(0.25, 0.5, 0.75), conf.int=FALSE)
25% 50% 75%
4.0 14.5 47.0
```

```
# Werte der geschätzten Survival-Funktion ohne Trennung nach Gruppen
> summary(KM0)
Call: survfit(formula = Surv(obsT, status) ~ 1, data = dfSurv,
  type = "kaplan-meier", conf.type = "log")

time n.risk n.event survival std.err lower 95% CI upper 95% CI
  1    180      17   0.9056  0.0218     0.8638   0.949
  2    163      12   0.8389  0.0274     0.7869   0.894
  3    151       9   0.7889  0.0304     0.7315   0.851
  4    142       9   0.7389  0.0327     0.6774   0.806
...

```

Die (hier gekürzte) Ausgabe nennt unter **time** die aufgetretenen Ereignis-Zeitpunkte t_i , unter **n.risk** die Anzahl der Personen, die vor t_i noch kein Ereignis hatten, unter **n.event** die Anzahl der Ereignisse zu t_i , unter **survival** die Schätzung $\hat{S}(t_i)$, unter **std.err** die geschätzte Streuung des Schätzers $\hat{S}(t_i)$ sowie in den letzten beiden Spalten die Grenzen des punktweisen Konfidenzintervalls für $\hat{S}(t_i)$.

Eine grafische Darstellung von $\hat{S}(t)$ inkl. der Konfidenzintervalle liefert `plot(<survfit-Objekt>)` (Abb. 9.2). Für die geschätzte kumulative Hazard-Funktion $\hat{\Lambda}(t) = -\ln \hat{S}(t)$ ist beim Aufruf von `plot()` das Argument `fun="cumhaz"` zu verwenden.

```
> plot(KM0, main=expression(paste("KM-Schätzer ", hat(S)(t),
+ " mit CI")), xlab="t", ylab="Survival", lwd=2)
```

```
> plot(KM, main=expression(paste("KM-Schätzer ", hat(S)[g](t), lty=1:3,
+ " für Gruppen")), xlab="t", ylab="Survival", lwd=2, col=1:3)

> legend(x="topright", lty=1:3, col=1:3, lwd=2, legend=LETTERS[1:3])

> plot(KMO, main=expression(paste("KM-Schätzer ", hat(Lambda)(t))),
+ xlab="t", ylab="kumulatives hazard", fun="cumhaz", lwd=2)
```

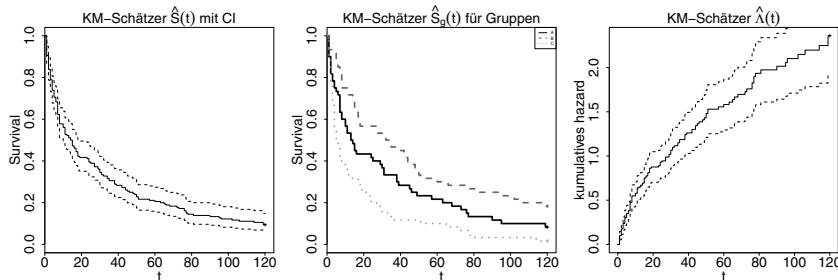


Abbildung 9.2: Kaplan-Meier-Schätzungen: Survival-Funktion $\hat{S}(t)$ ohne Berücksichtigung der Gruppen (mit Konfidenzintervallen), separate Schätzungen $\hat{S}_g(t)$ getrennt nach Gruppen g sowie die geschätzte kumulative Hazard-Funktion $\hat{\Lambda}(t)$ (mit Konfidenzintervallen)

9.3.2 Log-Rank-Test auf gleiche Survival-Funktionen

`survdiff()` aus dem Paket `survival` berechnet den Log-Rank-Test, ob sich die Survival-Funktionen in mehreren Gruppen unterscheiden.⁵

```
> survdiff(<Modellformel>, rho=0, data=<Datensatz>)
```

Als erstes Argument ist eine Modellformel der Form `<Surv-Objekt> ~ <Faktor>` zu übergeben. Stammen die dabei verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Das Argument `rho` kontrolliert, welche Testvariante berechnet wird. Mit der Voreinstellung 0 ist dies der Mantel-Hänszel-Test.

```
> library(survival) # für survdiff()
> survdiff(Surv(obsT, status) ~ IV, data=dfSurv)
Call:
survdiff(formula = Surv(obsT, status) ~ IV, data = dfSurv)
```

	N	Observed	Expected	$(O-E)^2/E$	$(O-E)^2/V$
IV=A	60	55	54.6	0.00351	0.00552
IV=B	60	49	72.3	7.50042	14.39151

⁵Für eine exakte Alternative vgl. `surv_test()` aus dem Paket `coin` (Hothorn, Hornik, van de Wiel & Zeileis, 2008).

IV=C 60 59 36.2 14.43764 20.10283

Chisq= 23.9 on 2 degrees of freedom, p= 6.55e-06

Die Ausgabe nennt in der Spalte N die jeweilige Gruppengröße, unter Observed die Anzahl der beobachteten Ereignisse f_g^o pro Gruppe g , unter Expected die dort unter der Nullhypothese erwartete Anzahl von Ereignissen f_g^e , unter $(O-E)^2/E$ den Wert für $\frac{(f_g^o - f_g^e)^2}{f_g^e}$ und unter $(O-E)^2/V \frac{(f_g^o - f_g^e)^2}{\hat{\sigma}_D^2}$, wobei $\hat{\sigma}_D$ die geschätzte Varianz von $D = f_g^o - f_g^e$ ist. Die letzte Zeile liefert den Wert der asymptotisch χ^2 -verteilten Teststatistik sowie die zugehörigen Freiheitsgrade mit dem p-Wert.

Bei geschichteten (stratifizierten) Stichproben lässt sich der Test auch unter Berücksichtigung der Schichtung durchführen. Dazu ist als weiterer Vorhersageterm auf der rechten Seite der Modellformel `strata(Faktor)` hinzuzufügen, wobei für dessen Gruppen keine Parameter geschätzt werden.

```
> survdiff(Surv(obsT, status) ~ IV + strata(sex), data=dfSurv)
Call:
survdiff(formula = Surv(obsT, status) ~ IV + strata(sex), data = dfSurv)
```

	N	Observed	Expected	$(O-E)^2/E$	$(O-E)^2/V$
IV=A	60	55	54.7	0.00185	0.00297
IV=B	60	49	72.1	7.42735	14.28570
IV=C	60	59	36.2	14.41091	20.32224

Chisq= 23.9 on 2 degrees of freedom, p= 6.45e-06

9.4 Cox proportional hazards Modell

Ein semi-parametrisches Regressionsmodell für censurierte Survival-Daten ist das Cox proportional hazards (PH) Modell, das den Einfluss von kontinuierlichen Prädiktoren ebenso wie von Gruppierungsfaktoren auf die Überlebenszeit einbeziehen kann. Das Modell lässt sich auf Daten anpassen, die aus unterschiedlichen Beobachtungszeiträumen hervorgegangen sind. Es macht keine spezifischen Voraussetzungen für die generelle Verteilungsform von Überlebenszeiten, basiert aber auf der Annahme, dass der Zusammenhang der p Prädiktoren X_j mit der logarithmierten Ereignisrate linear ist, sich also mit dem bekannten Regressionsmodell beschreiben lässt. Für die Form des Einflusses der X_j gelten insgesamt folgende Annahmen:

$$\begin{aligned} \ln \lambda(t) &= \ln \lambda_0(t) + \beta_1 X_1 + \cdots + \beta_p X_p &= \ln \lambda_0(t) + \mathbf{X}\boldsymbol{\beta} \\ \lambda(t) &= \lambda_0(t) e^{\beta_1 X_1 + \cdots + \beta_p X_p} &= \lambda_0(t) e^{\mathbf{X}\boldsymbol{\beta}} \\ S(t) &= S_0(t) \exp(\mathbf{X}\boldsymbol{\beta}) &= \exp(-\Lambda_0(t) e^{\mathbf{X}\boldsymbol{\beta}}) \\ \Lambda(t) &= \Lambda_0(t) e^{\mathbf{X}\boldsymbol{\beta}} \end{aligned}$$

Dabei spielt $\ln \lambda_0(t)$ die Rolle des absoluten Terms β_0 im GLM, entsprechend schließt die Abkürzung $\mathbf{X}\boldsymbol{\beta}$ keinen absoluten Term ein (vgl. Kap. 8). $\lambda_0(t)$ ist das baseline hazard, also die

für alle Beobachtungseinheiten identische Hazard-Funktion, wenn alle Einflussgrößen X_j gleich Null sind. Die Form dieser Funktion – und damit die generelle Verteilung von Überlebenszeiten (etwa Exponential- oder Weibull-Verteilung, vgl. Abschn. 9.1, 9.5) – bleibt unspezifiziert, weshalb dies der nonparametrische Teil des Modells ist.⁶

Ein exponenziertes Gewicht e^{β_j} ist der multiplikative Faktor, mit dem sich die Ereignisrate verändert, wenn der Prädiktor X_j um eine Einheit wächst. Dies ist das *hazard ratio*, also das relative hazard nach Erhöhung von X_j um eine Einheit zu vorher. Die prozentuale Veränderung der Ereignisrate ist $100(e^{\beta_j} - 1)$. Bei $e^{\beta_j} = 1.3$ wäre jede zusätzliche Einheit von X_j mit einer um 30% höheren Ereignisrate assoziiert, bei $e^{\beta_j} = 0.6$ mit einer um 40% niedrigeren Rate. Das Modell impliziert die Annahme, dass das relative hazard zweier Personen i und j mit Prädiktorwerten \mathbf{x}_i und \mathbf{x}_j unabhängig vom baseline hazard $\lambda_0(t)$ sowie unabhängig vom Zeitpunkt t konstant ist.

$$\frac{\lambda_i(t)}{\lambda_j(t)} = \frac{\lambda_0(t) e^{\mathbf{x}'_i \beta}}{\lambda_0(t) e^{\mathbf{x}'_j \beta}} = \frac{e^{\mathbf{x}'_i \beta}}{e^{\mathbf{x}'_j \beta}} = e^{(\mathbf{x}_i - \mathbf{x}_j)' \beta}$$

Das hazard von Person i ist demnach proportional zum hazard von Person j mit dem über die Zeit konstanten Proportionalitätsfaktor $e^{(\mathbf{x}_i - \mathbf{x}_j)' \beta}$. Diese Annahme proportionaler hazards von Personen mit unterschiedlichen Prädiktorwerten bedeutet, dass z. B. das hazard von Personen mit einer bestimmten Behandlungsmethode stets im selben Verhältnis zum hazard von Personen mit einer anderen Behandlungsmethode steht. Die hazards dürfen sich also nicht mit der Zeit annähern, stattdessen muss eine Methode gleichmäßig besser sein. Anders gesagt darf keine Interaktion $t \times \mathbf{X}$ vorliegen.

Das Cox PH-Modell wird mit `coxph()` aus dem Paket `survival` angepasst.

```
> coxph(<Modellformel>, ties="<Methode>", data=<Datensatz>)
```

Als erstes Argument ist eine Modellformel zu übergeben, deren linke Seite ein mit `Surv()` erstelltes Objekt ist (vgl. Abschn. 9.2). Die rechte Seite der Modellformel besteht aus kontinuierlichen Prädiktoren oder Faktoren, die zeitlich konstant oder – bei Daten in Zählprozess-Darstellung (vgl. Abschn. 9.2.2) – auch zeitabhängig sein können. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Zudem sind zwei besondere Vorhersageterme möglich: `strata(<Faktor>)` sorgt dafür, dass ein stratifiziertes Cox PH-Modell angepasst wird – mit einer separaten Baseline-Hazard-Funktion $\lambda_{0g}(t)$ für jede Stufe g des Faktors. Der Term `cluster` ist in Abschn. 9.4.5 beschrieben.

Das Modell berücksichtigt für die Anpassung die in Rangdaten transformierten, ggf. censierten Ereignis-Zeitpunkte, was es robust gegen Ausreißer macht. Gleichzeitig ist deshalb aber gesondert über das Argument `ties` zu spezifizieren, wie bei Bindungen, also identischen Ereignis-Zeitpunkten und damit uneindeutigen Rängen, vorzugehen ist. Voreinstellung ist "`efron`", für eine früher häufig gewählte Methode ist das Argument auf "`breslow`" zu setzen. Eine exakte, aber rechenintensive Behandlung von Bindungen erhält man mit "`exact`".

Das Cox PH-Modell soll hier für die in Abschn. 9.2.1 simulierten Daten mit zeitlich konstanten Prädiktoren und höchstens einmal auftretenden Ereignissen angepasst werden.

⁶Aus diesem Grund ist der Aussagebereich eines angepassten Cox PH-Modells auch auf die in der Stichprobe tatsächlich vorliegenden Überlebenszeiten begrenzt, eine Extrapolation über die maximal beobachtete Überlebenszeit hinaus also unzulässig.

```
> library(survival) # für coxph()
> (fitCPH <- coxph(Surv(obsT, status) ~ X + IV, data=dfSurv))
Call:
coxph(formula = Surv(obsT, status) ~ X + IV, data = dfSurv)

      coef exp(coef) se(coef)     z      p
X    0.491    1.634   0.0869  5.66 1.5e-08
IVB -0.406    0.666   0.1968 -2.06 3.9e-02
IVC  0.579    1.784   0.1912  3.03 2.5e-03

Likelihood ratio test=56  on 3 df, p=4.1e-12  n= 180, number of events= 163
```

äquivalenter Aufruf mit Daten in Zählprozess-Darstellung

```
> coxph(Surv(start, obsT, status) ~ X + IV, data=dfSurvCP) # ...
```

Die Ausgabe nennt in der Spalte `coef` die geschätzten Koeffizienten $\hat{\beta}_j$, deren exponenzierte Werte $e^{\hat{\beta}_j}$ in der Spalte `exp(coef)` stehen. Für den kontinuierlichen Prädiktor X_j ist $e^{\hat{\beta}_j}$ der Änderungsfaktor für die geschätzte Ereignisrate $\hat{\lambda}(t)$ (also das hazard ratio), wenn X_j um eine Einheit wächst. Die Zeilen IVB und IVC sind so zu interpretieren, dass A als erste Stufe des Faktors IV als Referenzgruppe verwendet wurde (vgl. Abschn. 2.6.5), so dass Dummy-codierte Variablen für die Stufen B und C verbleiben (Treatment-Kontraste, vgl. Abschn. 12.9.2). Für IVB und IVC ist $e^{\hat{\beta}_j}$ daher jeweils der multiplikative Änderungsfaktor für $\hat{\lambda}(t)$ verglichen mit Gruppe A. Die geschätzten Streuungen $\hat{\sigma}_{\hat{\beta}}$ der $\hat{\beta}_j$ stehen in der Spalte `se(coef)`, die Werte der Wald-Statistik $z = \frac{\hat{\beta}}{\hat{\sigma}_{\hat{\beta}}}$ in der Spalte `z` und die zugehörigen p -Werte in der Spalte `p`. Dieser Wald-Test setzt voraus, dass z unter der Nullhypothese asymptotisch standardnormalverteilt ist.

Die letzte Zeile berichtet die Ergebnisse des Likelihood-Quotienten-Tests des Gesamtmodells gegen das Modell ohne Prädiktoren. Teststatistik ist die Devianz-Differenz beider Modelle mit der Differenz ihrer Freiheitsgrade als Anzahl der Freiheitsgrade der asymptotisch gültigen χ^2 -Verteilung. Zusätzliche Informationen liefert `summary(<coxph-Objekt>)`.

```
> summary(fitCPH)
Call:
coxph(formula = Surv(obsT, status) ~ X + IV, data = dfSurv)
```

n= 180, number of events= 163

	coef	exp(coef)	se(coef)	z	Pr(> z)
X	0.49123	1.63433	0.08685	5.656	1.55e-08 ***
IVB	-0.40612	0.66623	0.19675	-2.064	0.03901 *
IVC	0.57890	1.78407	0.19117	3.028	0.00246 **

	exp(coef)	exp(-coef)	lower .95	upper .95
X	1.6343	0.6119	1.3785	1.9376
IVB	0.6662	1.5010	0.4531	0.9797
IVC	1.7841	0.5605	1.2266	2.5950

```
Concordance= 0.683  (se = 0.027 )
Rsquare= 0.268  (max possible= 1 )
Likelihood ratio test= 56.05  on 3 df,  p=4.103e-12
Wald test            = 54.98  on 3 df,  p=6.945e-12
Score (logrank) test = 54.86  on 3 df,  p=7.365e-12
```

Neben den bereits erwähnten Informationen enthält die Ausgabe zusätzlich die Konfidenzintervalle für die exponenzierten Schätzungen $e^{\hat{\beta}_j}$ in den Spalten `lower .95` und `upper .95`. Die Konkordanz ist der Anteil an allen Paaren von Beobachtungsobjekten, bei denen das Beobachtungsobjekt mit empirisch kürzerer Überlebenszeit auch ein höheres vorhergesagtes hazard besitzt. Liegen keine Bindungen vor, ist die Konkordanz daher gleich Kendalls τ von T und $\hat{\lambda}$ (vgl. Abschn. 10.3.1). Der unter `Rsquare` angegebene pseudo- R^2 -Wert ist jener nach Cox & Snell (vgl. Abschn. 8.1.3). Die Ergebnisse des Likelihood-Quotienten-, Wald- und Score-Tests beziehen sich alle auf den Test des Gesamtmodells gegen jenes ohne Prädiktoren.

9.4.1 Anpassungsgüte und Modelltests

Aus einem von `coxph()` zurückgegebenen Objekt lassen sich weitere Informationen zur Anpassungsgüte extrahieren, darunter der Wert des Informationskriteriums AIC mit `extractAIC()` oder die pseudo- R^2 -Werte nach McFadden, Cox & Snell und Nagelkerke (vgl. Abschn. 8.1.3). Dafür enthält das Objekt in der Komponente `loglik` einen Vektor mit den maximierten geschätzten likelihoods des Modells ohne Prädiktoren und des angepassten Modells.

```
> library(survival)                                # für coxph()
> extractAIC(fitCPH)                             # AIC
[1] 3.000 1399.438

> LL0 <- fitCPH$loglik[1]                      # log-likelihood 0-Modell
> LLf <- fitCPH$loglik[2]                       # log-likelihood angepasstes Modell

> as.vector( 1 - (LLf / LL0))                  # R^2 McFadden
[1] 0.03866744

> as.vector( 1 - exp((2/N) * (LL0 - LLf)))    # R^2 Cox & Snell
[1] 0.2675625

# R^2 Nagelkerke
> as.vector((1 - exp((2/N) * (LL0 - LLf))) / (1 - exp(LL0^(2/N))))
[1] 0.2676477
```

Da der hier im Modell berücksichtigte Faktor IV mit mehreren Parametern β_j assoziiert ist, muss seine Signifikanz insgesamt über einen Modellvergleich getestet werden. Dazu dient ein Likelihood-Quotienten-Test, der auf der asymptotisch χ^2 -verteilten Devianz-Differenz zweier hierarchischer Modelle mit demselben Kriterium beruht (vgl. Abschn. 8.1.5): Der Prädiktorensatz des eingeschränkten Modells (`fitR`) ist dabei vollständig im Prädiktorensatz des umfassenderen

Modells `<fitU>` enthalten, das zusätzlich noch den Faktor berücksichtigt. Der Test erfolgt dann mit `anova(<fitR>, <fitU>)`.

```
# eingeschränktes Modell ohne Faktor IV
> fitCPH1 <- coxph(Surv(obsT, status) ~ X, data=dfSurv)
> anova(fitCPH1, fitCPH) # LQ-Modelltest für IV
Analysis of Deviance Table
Cox model: response is Surv(obsT, status)
Model 1: ~ X
Model 2: ~ X + IV
  loglik  Chisq Df P(>|Chis|)
1 -708.98
2 -696.72 24.52  2 4.738e-06 ***
```

9.4.2 Survival-Funktion und baseline hazard schätzen

Analog zur Kaplan-Meier-Analyse (vgl. Abschn. 9.3.1) schätzt `survfit(<coxph-Objekt>)` die Survival-Funktion $\hat{S}_{\bar{x}}(t)$ im Cox PH-Modell für ein pseudo-Beobachtungsobjekt, das als Prädiktorwerte den jeweiligen Mittelwert für die gegebene Stichprobe besitzt (kurz: \bar{x}). Der Median der geschätzten Überlebenszeit findet sich in der Ausgabe unter `median`, beliebige Quantile von $\hat{S}_{\bar{x}}(t)$ erhält man mit `quantile(<coxph-Objekt>, probs=(Quantile))`.

```
> library(survival) # für survfit(), basehaz()
> (CPH <- survfit(fitCPH, conf.type="log"))
Call: survfit(formula = fitCPH)

records n.max n.start events median 0.95LCL 0.95UCL
      180     180      180     163      15       11       19

# Quantile der Überlebenszeit ohne Grenzen der Konfidenzintervalle
> quantile(CPH, probs=c(0.25, 0.5, 0.75), conf.int=FALSE)
25% 50% 75%
  5   15   42
```

Berücksichtigt das Modell Faktoren, ist die mittlere pseudo-Beobachtung \bar{x} kaum zu interpretieren, da für sie die Mittelwerte der dichotomen Indikatorvariablen gebildet werden (vgl. Abschn. 12.9.2). Diese Mittelwerte entsprechen damit keiner tatsächlich vorhandenen Gruppe. Oft ist es dann angemessener, an das Argument `newdata` von `survfit()` einen Datensatz zu übergeben, der neue Daten für Variablen mit denselben Namen, und bei Faktoren zusätzlich denselben Stufen wie jene der ursprünglichen Prädiktoren enthält. In diesem Fall berechnet `survfit()` für jede Zeile des Datensatzes die Schätzung $\hat{S}(t)$. Auf diese Weise kann $\hat{S}(t)$ etwa für bestimmte Gruppenzugehörigkeiten oder Werte anderer Prädiktoren ermittelt werden. In der von `survfit()` zurückgegebenen Liste stehen die Werte für t in der Komponente `time`, jene für $\hat{S}(t)$ in der Komponente `surv`. Dabei ist `surv` eine Matrix mit so vielen Zeilen, wie es Werte für t gibt und so vielen Spalten, wie neue Beobachtungsobjekte (Zeilen von `newdata`) vorhanden sind.

```
# Datensatz: 2 Frauen mit Prädiktor X=-2 in Gruppe A bzw. in C
> dfNew <- data.frame(sex=factor(c("f", "f")), levels=levels(dfSurv$sex)),
+                         X=c(-2, -2),
+                         IV=factor(c("A", "C"), levels=levels(dfSurv$IV)))

# wende angepasstes Cox PH-Modell auf neue Daten an
> CPHnew <- survfit(fitCPH, newdata=dfNew)
```

Die grafische Darstellung von $\hat{S}_{\bar{x}}(t)$ bzw. von $\hat{S}(t)$ für die Beobachtungen in `newdata` erfolgt mit `plot(<survfit-Objekt>)` (Abb. 9.3). Für die geschätzte kumulative Hazard-Funktion $\hat{\Lambda}(t) = -\ln \hat{S}(t)$ ist beim Aufruf von `plot()` das Argument `fun="cumhaz"` zu verwenden.

```
# Darstellung geschätztes S(t) für mittlere pseudo-Beobachtung
> plot(CPH, main=expression(paste("Cox PH-Schätzung ", hat(S)(t),
+ " mit CI")), xlab="t", ylab="Survival", lwd=2)

# füge geschätztes S(t) für neue Daten hinzu
> lines(CPHnew$time, CPHnew$surv[, 1], lwd=2, col="blue")
> lines(CPHnew$time, CPHnew$surv[, 2], lwd=2, col="red")
> legend(x="topright", lwd=2, col=c("black", "blue", "red"),
+         legend=c("pseudo-Beobachtung", "sex=f, X=-2, IV=A",
+                 "sex=f, X=-2, IV=C"))
```

Das geschätzte kumulative baseline hazard erhält man mit `basehaz(<coxph-Objekt>)` aus dem Paket `survival` (Abb. 9.3). Die Schätzung $\hat{\Lambda}_{\bar{x}}(t)$ kann dabei mit dem Argument `centered=TRUE` für ein pseudo-Beobachtungsobjekt berechnet werden, das für die vorliegende Stichprobe gemittelte Prädiktorwerte \bar{x} besitzt. Setzt man dagegen `centered=FALSE`, bezieht sich das Ergebnis $\hat{\Lambda}_0(t)$ i. S. einer echten baseline bei Treatment-Kontrasten auf ein Beobachtungsobjekt in der Referenzgruppe eines Faktors, für das alle Prädiktorwerte gleich 0 sind. Das Ergebnis ist ein Datensatz mit den Variablen `hazard` für $\hat{\Lambda}_0(t)$ und `time` für t .

Um das geschätzte kumulative hazard $\hat{\Lambda}(t)$ für beliebige Werte der Prädiktoren zu ermitteln, ist $\hat{\Lambda}_0(t) \cdot e^{X\hat{\beta}}$ zu bilden. Soll $\hat{\Lambda}(t)$ nicht für die Referenzgruppe, sondern für eine andere Stufe j eines Faktors berechnet werden, vereinfacht sich der Ausdruck bei Treatment-Kontrasten zu $\hat{\Lambda}_0(t) \cdot e^{\hat{\beta}_j}$.

```
# exponentisierte geschätzte Koeffizienten = Faktoren für hazard
> expCoef <- exp(coef(fitCPH))

# kumulatives hazard für Referenzgruppe A und X=0
> Lambda0A <- basehaz(fitCPH, centered=FALSE)
> Lambda0B <- expCoef[2]*Lambda0A$hazard          # kumulatives hazard B
> Lambda0C <- expCoef[3]*Lambda0A$hazard          # kumulatives hazard C

# stelle kumulatives hazard für Gruppe A dar
> plot(hazard ~ time, main=expression(paste("Cox PH-Schätzung ",
+     hat(Lambda)[g](t), " pro Gruppe")), type="s", ylim=c(0, 5),
+     xlab="t", ylab="kumulatives hazard", lwd=2, data=Lambda0A)
```

```
# füge kumulatives hazard für Gruppe B und C hinzu
> lines(Lambda0A$time, Lambda0B, lwd=2, col="red")
> lines(Lambda0A$time, Lambda0C, lwd=2, col="green")
> legend(x="bottomright", lwd=2, col=1:3, legend=LETTERS[1:3])
```

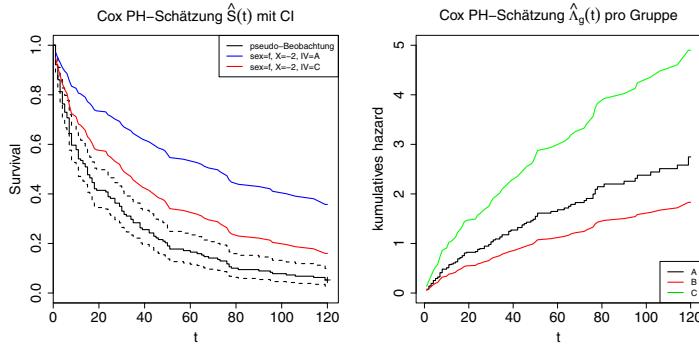


Abbildung 9.3: Cox PH-Schätzungen: Survival-Funktion $\hat{S}_{\bar{x}}(t)$ mit Konfidenzintervallen für ein pseudo-Beobachtungsobjekt mit mittleren Prädiktorwerten \bar{x} sowie kumulative Hazard-Funktion $\hat{\Lambda}_g(t)$ getrennt nach Gruppen g

9.4.3 Modelldiagnostik

Um die Angemessenheit des Cox PH-Modells für die gegebenen Daten beurteilen zu können, sollten drei Aspekte der Modellanpassung geprüft werden: Die Annahme proportionaler hazards, das Vorhandensein besonders einflussreicher Beobachtungen sowie die Linearität von $\ln \lambda(t)$ bzgl. der kontinuierlichen Prädiktoren.

Aus der Annahme proportionaler hazards folgt, dass $\ln(-\ln(S(t)))$ linear mit $\ln t$ ist. Diese Voraussetzung lässt sich in einem Diagramm prüfen, das $\ln(-\ln(\hat{S}_g(t_i)))$ gegen $\ln t_i$ aufträgt, wobei $\hat{S}_g(t_i)$ die geschätzten Kaplan-Meier Survival-Funktionen der Überlebenszeit für die Stufen g eines Faktors sind (Abb. 9.4). Dafür ist separat für jeden Prädiktor ein Kaplan-Meier Modell anzupassen und mit `plot(KM-Modell, fun="cloglog")` darzustellen. Damit sich die PH-Annahme bzgl. kontinuierlicher Prädiktoren auf diese Weise prüfen lässt, müssen diese zunächst in Gruppen eingeteilt werden (vgl. Abschn. 2.6.7).

Die Survival-Funktionen sollten im Diagramm linear mit $\ln t_i$ ansteigen und zudem parallel verlaufen. Im Spezialfall des Modells, das für T eine Exponentialverteilung annimmt (vgl. Abschn. 9.5), sollte die Steigung von $\hat{S}_g(t_i)$ gleich 1 sein. Ist die PH-Annahme offensichtlich für einen Prädiktor verletzt, besteht eine Strategie darin, bzgl. dieses Prädiktors zu stratifizieren – bei kontinuierlichen Variablen nach Einteilung in geeignete Gruppen.

```
> library(survival) # für survfit(), cox.zph()
# teile X per Median-Split in zwei Gruppen
```

```

> dfSurv <- transform(dfSurv, Xcut=cut(X, breaks=c(-Inf, median(X), Inf)))
> KMiv   <- survfit(Surv(obsT, status) ~ IV,      # KM-Schätzungen für IV
+                      type="kaplan-meier", data=dfSurv)

> KMxcut <- survfit(Surv(obsT, status) ~ Xcut,    # KM-Schätzungen für X
+                      type="kaplan-meier", data=dfSurv)

# Diagramme  $\ln(-\ln(S(t)))$  gegen  $\ln t$ 
> plot(KMiv, fun="cloglog", main="cloglog-Plot für IV1",
+       xlab="ln t", ylab=expression(ln(-ln(hat(S)[g](t)))), 
+       col=c("black", "blue", "red"), lty=1:3)

> legend(x="topleft", col=c("black", "blue", "red"), lwd=2,
+         lty=1:3, legend=LETTERS[1:3])

> plot(KMxcut, fun="cloglog", main="cloglog-Plot für Xcut",
+       xlab="ln t", ylab=expression(ln(-ln(hat(S)[g](t)))), 
+       col=c("black", "blue"), lty=1:2)

> legend(x="topleft", col=c("black", "blue"), lwd=2,
+         legend=c("lo", "hi"), lty=1:2)

```

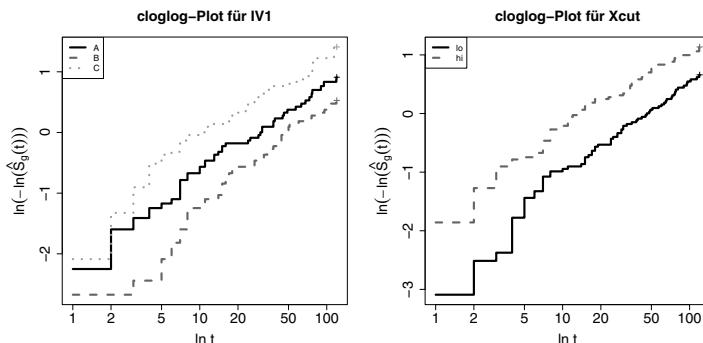


Abbildung 9.4: Beurteilung der Annahme proportionaler hazards bzgl. jedes Prädiktors anhand der Linearität von $\ln(-\ln(\hat{S}_g(t_i)))$ mit $\ln t_i$

Ähnlich wie in der Regressionsdiagnostik (vgl. Abschn. 6.5) kann sich die Beurteilung der Voraussetzungen des Cox PH-Modells auch auf die Verteilung von Residuen stützen, insbesondere auf die Schönfeld- und Martingal-Residuen. Beide erhält man mit `residuals(<coxph-Objekt>, type="<Typ>")`, wobei `type` auf "scaledsch" bzw. auf "martingale" zu setzen ist. Das Ergebnis ist eine Matrix, die für jede Beobachtung (Zeilen) das Residuum bzgl. jedes Prädiktors (Spalten) enthält.

`cox.zph(<coxph-Objekt>)` berechnet auf Basis der Schönfeld-Residuen für jeden Prädiktor sowie für das Gesamtmodell einen Test der Nullhypothese, dass die Annahme proportionaler

hazards stimmt.

```
> (czph <- cox.zph(fitCPH))
      rho chisq   p
X     -0.0959 1.5316 0.216
IVB    0.1001 1.6013 0.206
IVC    0.0216 0.0761 0.783
GLOBAL      NA 3.2264 0.358
```

Das von `cox.zph()` ausgegebene Objekt lässt sich an `plot()` übergeben, um die Schönfeld-Residuen für jeden Prädiktor gegen eine Transformation der Überlebenszeit darzustellen (Abb. 9.5). Die Diagramme enthalten zur Verdeutlichung des Verlaufs eine Spline-Interpolation (vgl. Abschn. 14.7.2) inkl. des zugehörigen Bereichs von ± 2 Standardfehlern. Gibt es eine systematische Variation der Residuen in Abhängigkeit von der Überlebenszeit, ist das ein Hinweis darauf, dass die Annahme proportionaler hazards verletzt ist.

```
> par(mfrow=c(2, 2))                                # Platz für 4 panels
> plot(czph)                                         # Residuen und splines
```

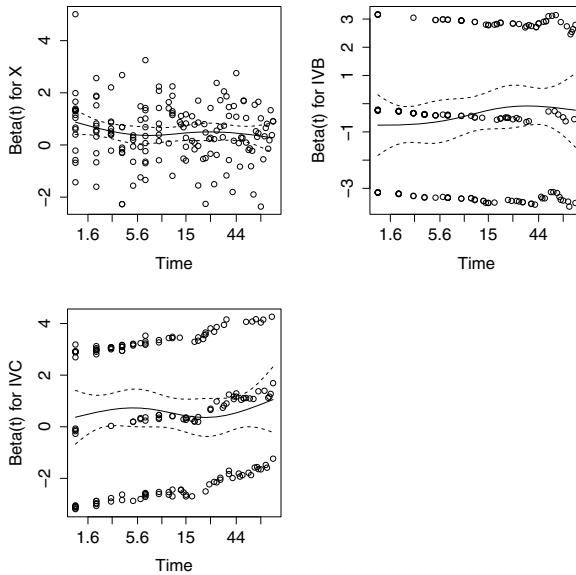


Abbildung 9.5: Beurteilung der Annahme proportionaler hazards bzgl. jedes Prädiktors anhand skalarierter Schönfeld-Residuen

Einflussreiche Beobachtungen können ähnlich wie in der linearen Regression über die von ihnen verursachten Änderungen in den geschätzten Parametern $\hat{\beta}_j$ diagnostiziert werden (vgl. Abschn. 6.5.1). Das standardisierte Maß DFBETAS erhält man für jede Beobachtung und jeden Prädiktor, indem man im Aufruf von `residuals()` das Argument `type="dfbetas"` wählt (Abb. 9.6).

```
# Matrix der standardisierten Einflussgrößen DfBETAS
> dfbetas <- residuals(fitCPH, type="dfbetas")
> plot(dfbetas[ , 1], type="h", main="DfBETAS für X",
+       ylab="DfBETAS", lwd=2)

> plot(dfbetas[ , 2], type="h", main="DfBETAS für IV-B",
+       ylab="DfBETAS", lwd=2)

> plot(dfbetas[ , 3], type="h", main="DfBETAS für IV-C",
+       ylab="DfBETAS", lwd=2)
```

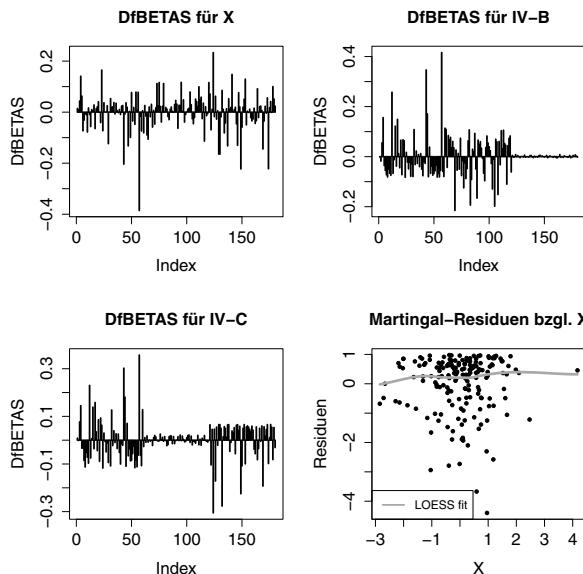


Abbildung 9.6: Diagnostik einflussreicher Beobachtungen anhand der DfBETAS-Werte für jeden Prädiktor sowie Beurteilung der Linearität bzgl. des kontinuierlichen Prädiktors

Laut Modell sollte der Zusammenhang von $\ln \lambda(t)$ mit den Prädiktoren X_j linear sein. Inwieweit die Daten mit dieser Annahme konsistent sind, lässt sich über den Verlauf der Martingal-Residuen in Abhängigkeit von den Werten der kontinuierlichen X_j einschätzen (Abb. 9.6). Zur grafischen Beurteilung der Verteilung ist es dabei hilfreich, einen nonparametrischen LOESS-Glättter (vgl. Abschn. 14.7.1) einzuziehen, der horizontal verlaufen sollte.

```
> resMart <- residuals(fitCPH, type="martingale")
> plot(dfSurv$X, resMart, main="Martingal-Residuen bzgl. X",
+       xlab="X", ylab="Residuen", pch=20)

# zeichne zusätzlich LOESS-Glättter ein
> lines(loess.smooth(dfSurv$X, resMart), lwd=2, col="blue")
```

```
> legend(x="bottomleft", col="blue", lwd=2, legend="LOESS fit")
```

9.4.4 Vorhersage und Anwendung auf neue Daten

Wie im GLM liefert `predict(<coxph-Modell>, type="<Typ>")` für jedes Beobachtungsobjekt i Schätzungen verschiedener Kennwerte. Mit dem Argument `type="risk"` erhält man die hazard ratios $e^{(\mathbf{x}_i - \bar{\mathbf{x}})^\hat{\beta}}$ zu einem pseudo-Beobachtungsobjekt, das als Prädiktorwerte den jeweiligen Mittelwert jedes Prädiktors aus der Stichprobe besitzt (kurz: $\bar{\mathbf{x}}$, vgl. Abschn. 9.4.2). Für stratifizierte Modelle werden diese Mittelwerte dabei pro Schicht gebildet, es sei denn man setzt das Argument `reference="sample"`. Für die Werte des linearen Prädiktors $\mathbf{X}\hat{\beta}$ selbst ist `type="lp"` zu verwenden.

Zusätzlich akzeptiert das Argument `newdata` von `predict()` einen Datensatz, der neue Daten für Variablen mit denselben Namen, und bei Faktoren zusätzlich denselben Stufen wie jene der ursprünglichen Prädiktoren enthält. Als Ergebnis erhält man die vorhergesagten hazard ratios für die neuen Prädiktorwerte (vgl. Abschn. 6.4).

`survexp()` aus dem Paket `survival` ermittelt auf Basis eines Cox PH-Modells die Vorhersage der Survival-Funktion $\hat{S}(t_i)$ in Abhängigkeit von Prädiktorwerten.

```
> survexp(<Modellformel>, ratetable="<coxph-Modell>", data=<Datensatz>)
```

Als erstes Argument ist die rechte Seite einer Modellformel mit den Prädiktoren anzugeben, für deren Kombination von Werten jeweils der Wert der Survival-Funktion geschätzt werden soll. Als Basis dient ein Cox PH-Modell, das für `ratetable` zu übergeben ist. Dieses Modell liefert auch die Ereigniszeiten t_i , für die $\hat{S}(t_i)$ bestimmt wird. Der Datensatz mit Daten für die Variablen der Modellformel wird von `data` akzeptiert.

Die zurückgegebene Liste enthält in der Komponente `time` die Ereigniszeiten t_i und in der Komponente `surv` die Werte von $\hat{S}(t_i)$. Ohne Trennung nach Gruppen oder anderen Prädiktoren (Modellformel ~ 1) ist `surv` ein Vektor. Stehen Prädiktoren in der Modellformel ($\sim \mathbf{X} + \text{IV1} \rightarrow + \dots$), ist `surv` eine Matrix mit je einer Spalte für jede Kombination g von Faktorstufen und Werten der kontinuierlichen Prädiktoren. Die zugehörigen Werte von $\hat{S}_g(t_i)$ stehen in den Zeilen.

```
# Vorhersage für Gesamtgruppe ohne Trennung nach Prädiktoren
> Shat1 <- survexp(~ 1, ratetable=fitCPH, data=dfSurv)
> with(Shat1, head(data.frame(time, surv), n=4))
  time      surv
1   1  0.9053526
2   2  0.8372556
3   3  0.7858015
4   4  0.7342081

# Vorhersage getrennt nach Gruppen von IV
> Shat2 <- survexp(~ IV, ratetable=fitCPH, data=dfSurv)
> with(Shat2, head(data.frame(time, surv), n=4))
  time    IV.A    IV.B    IV.C
```

```

1 1 0.9195093 0.9455425 0.8510060
2 2 0.8582419 0.9028127 0.7507122
3 3 0.8104261 0.8686281 0.6783505
4 4 0.7613120 0.8326912 0.6086210

```

9.4.5 Erweiterungen des Cox PH-Modells

Das Cox PH-Modell lässt sich auf verschiedene Situationen erweitern:

- Zeitvariierende Kovariaten X können über die Zählprozess-Darstellung von Daten (vgl. Abschn. 9.2.2) repräsentiert und etwa in Interaktionstermen $X \times t$ in die Modellformel von `coxph()` aufgenommen werden. Dafür ist es notwendig, die Gesamt-Beobachtungsintervalle aller Personen mit `survSplit()` an allen vorkommenden Ereigniszeiten in Teilintervalle zu zerlegen.
- Pro Beobachtungsobjekt potentiell mehrfach auftretende Ereignisse lassen sich ebenfalls in Zählprozess-Darstellung speichern. In der Modellformel von `coxph()` kann dann ein Vorhersageterm `cluster(<ID>)` hinzugefügt werden, wobei der Faktor `<ID>` codiert, von welchem Beobachtungsobjekt ein Intervall stammt. Dies bewirkt eine robuste Schätzung der Kovarianzmatrix der Parameterschätzer. Alternativ können wiederkehrende Ereignisse durch Stratifizierung analysiert werden, wobei die Beobachtungen mit jeweils dem ersten, zweiten, dritten, ... Ereignis ein Stratum bilden.
- Penalisierte Cox-Modelle können mit der Funktion `coxnet()` aus dem Paket `glmnet` (vgl. Abschn. 6.6.2) sowie mit dem Paket `coxphf` (Ploner & Heinze, 2013) angepasst werden.
- Für Hinweise auf Pakete zur Auswertung mit *frailty* oder *competing risks* Modellen vgl. den Abschnitt *Survival Analysis* der CRAN Task Views (Allignol & Latouche, 2014).

9.5 Parametrische proportional hazards Modelle

Bei spezifischen Vorstellungen über die Verteilung der Überlebenszeit T kommen auch parametrische Regressionsmodelle in Betracht, die sich unter Beibehaltung der Annahme proportionaler hazards als Spezialfälle des Cox PH-Modells ergeben (vgl. Abschn. 9.4). Für exponential- oder Weibull-verteilte Überlebenszeiten gibt es dabei zwei äquivalente Möglichkeiten, das lineare Regressionsmodell zu formulieren: Zum einen wie im Cox PH-Modell für das logarithmierte hazard, zum anderen für die logarithmierte Überlebenszeit. Bei der zweiten Darstellung spricht man von einem *accelerated failure time* Modell (AFT).

9.5.1 Darstellung über die Hazard-Funktion

Spezialfälle des Cox PH-Modells ergeben sich, wenn für das baseline hazard $\lambda_0(t)$ eine Verteilung angenommen wird, die mit einer Exponential- oder Weibull-Verteilung von T korrespondiert. Die logarithmierte Ereignisrate soll wie im Cox PH-Modell linear von den Prädiktoren X_j abhängen. Das baseline hazard $\lambda_0(t)$ ist dabei der Verlauf der Ereignisrate für ein Beobachtungsobjekt,

für das alle X_j gleich 0 sind. Die exponenzierten Parameter e^{β_j} geben wie im Cox PH-Modell den Änderungsfaktor für die Ereignisrate (also das hazard ratio) an, wenn ein Prädiktor um 1 wächst.

Bei Annahme einer Exponentialverteilung von T mit Erwartungswert $E(T) = b > 0$ und Varianz b^2 ergibt sich die Dichtefunktion $f(t) = \lambda(t) S(t)$ aus der konstanten Hazard-Funktion $\lambda(t) = \lambda = \frac{1}{b}$ und der Survival-Funktion $S(t) = e^{-\frac{t}{b}}$. Die kumulative Hazard-Funktion ist $\Lambda(t) = \frac{t}{b}$. Oft wird die Exponentialverteilung auch mit der Grundrate λ als $f(t) = \lambda e^{-\lambda t}$ bzw. $S(t) = e^{-\lambda t}$ und $\Lambda(t) = \lambda t$ formuliert. In `rexp()` ist mit dem Argument `rate` λ gemeint. Durch den Einfluss der X_j ergibt sich dann als neue Grundrate $\lambda' = \lambda e^{X\beta}$. Insgesamt resultieren aus der Spezialisierung des Cox PH-Modells folgende Modellvorstellungen, wobei die Abkürzung $X\beta$ keinen absoluten Term β_0 einschließt:

$$\begin{aligned}\lambda(t) &= \frac{1}{b} e^{X\beta} &= \lambda e^{X\beta} \\ \ln \lambda(t) &= -\ln b + X\beta &= \ln \lambda + X\beta \\ S(t) &= \exp\left(-\frac{t}{b} e^{X\beta}\right) &= \exp\left(-\lambda t e^{X\beta}\right) \\ \Lambda(t) &= \frac{t}{b} e^{X\beta} &= \lambda t e^{X\beta}\end{aligned}$$

Die Dichtefunktion einer Weibull-Verteilung kann unterschiedlich formuliert werden. `rweibull()` verwendet den Formparameter $a > 0$ für das Argument `shape` und den Skalierungsparameter $b > 0$ für `scale`. Für $a > 1$ steigt das hazard mit t , für $a < 1$ sinkt es, und für $a = 1$ ist es konstant. Die Exponentialverteilung ist also ein Spezialfall der Weibull-Verteilung für $a = 1$. b ist die *charakteristische Lebensdauer*, nach der $1 - \frac{1}{e} \approx 63.2\%$ der Ereignisse aufgetreten sind ($S(b) = \frac{1}{e}$). Die Dichtefunktion $f(t) = \lambda(t) S(t)$ ergibt sich mit dieser Wahl aus der Hazard-Funktion $\lambda(t) = \frac{a}{b} \left(\frac{t}{b}\right)^{a-1}$ und der Survival-Funktion $S(t) = \exp(-(\frac{t}{b})^a)$. Die kumulative Hazard-Funktion ist $\Lambda(t) = (\frac{t}{b})^a$ mit der Umkehrfunktion $\Lambda^{-1}(t) = (b t)^{\frac{1}{a}}$. Der Erwartungswert ist $E(T) = b \Gamma(1 + \frac{1}{a})$.

Analog zur Exponentialverteilung lässt sich die Weibull-Verteilung auch mit $\lambda = \frac{1}{b^a}$ formulieren, so dass $\lambda(t) = \lambda a t^{a-1}$, $S(t) = \exp(-\lambda t^a)$ und $\Lambda(t) = \lambda t^a$ gilt. Durch den Einfluss der X_j ergibt sich dann $\lambda' = \lambda e^{X\beta}$. Insgesamt impliziert das Weibull-Modell folgende Zusammenhänge:

$$\begin{aligned}\lambda(t) &= \frac{a}{b} \left(\frac{t}{b}\right)^{a-1} e^{X\beta} &= \lambda a t^{a-1} e^{X\beta} \\ \ln \lambda(t) &= \ln \left(\frac{a}{b} \left(\frac{t}{b}\right)^{a-1}\right) + X\beta &= \ln \lambda + \ln a + (a-1) \ln t + X\beta \\ S(t) &= \exp\left(-\left(\frac{t}{b}\right)^a e^{X\beta}\right) &= \exp\left(-\lambda t^a e^{X\beta}\right) \\ \Lambda(t) &= \left(\frac{t}{b}\right)^a e^{X\beta} &= \lambda t^a e^{X\beta}\end{aligned}$$

9.5.2 Darstellung als accelerated failure time Modell

Das betrachtete Exponential- und Weibull-Modell lässt sich äquivalent auch jeweils als lineares Modell der logarithmierten Überlebenszeit formulieren (accelerated failure time Modell, AFT).

$$\ln T = X\gamma + z = \mu + \sigma\epsilon$$

Dabei ist ϵ ein Fehlerterm, der im Weibull-Modell einer Typ-I (Gumbel) Extremwertverteilung folgt und durch $\sigma = \frac{1}{a}$ skaliert wird. Sind t_i zufällige Überlebenszeiten aus einer

Weibull-Verteilung, sind damit $\ln t_i$ zufällige Beobachtungen einer Extremwertverteilung mit Erwartungswert $\mu = \ln b$. Mit $a = 1$ ergibt sich als Spezialfall das Exponential-Modell.

Ein Parameter γ_j ist im AFT-Modell das über t konstante Verhältnis zweier Quantile von $S(t)$, wenn sich der Prädiktor X_j um eine Einheit erhöht. Ein exponenzierteter Parameter e^{γ_j} gibt analog den Änderungsfaktor für die Überlebenszeit bei einer Änderung von X_j um eine Einheit an. Zwischen dem Parameter β_j in der Darstellung als PH-Modell und dem Parameter γ_j in der Formulierung als AFT-Modell besteht für Weibull-verteilte Überlebenszeiten die Beziehung $\beta_j = -\frac{\gamma_j}{a}$, für den Spezialfall exponentialverteilter Überlebenszeiten also $\beta_j = -\gamma_j$.

9.5.3 Anpassung und Modelltests

AFT-Modelle können mit `survreg()` aus dem Paket `survival` angepasst werden.

```
> survreg(<Modellformel>, dist="<Verteilung>", data=<Datensatz>)
```

Als erstes Argument ist eine Modellformel zu übergeben, deren linke Seite ein mit `Surv()` erstelltes Objekt ist (vgl. Abschn. 9.2). Dabei ist sicherzustellen, dass alle Ereignis-Zeitpunkte $t_i > 0$ sind und nicht (etwa durch Rundung) Nullen enthalten. Die rechte Seite der Modellformel kann neben – zeitlich konstanten – kontinuierlichen Prädiktoren und Faktoren als besonderen Vorhersageterm `strata(<Faktor>)` umfassen. Dieser sorgt dafür, dass ein stratifiziertes Modell angepasst wird, das eine separate Baseline-Hazard-Funktion $\lambda_{0g}(t)$ für jede Stufe g des Faktors beinhaltet. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Das Argument `dist` bestimmt die für T angenommene Verteilung – mögliche Werte sind etwa "weibull" oder "exponential" (für weitere vgl. `?survreg`).

Die Modelle sollen hier für die in Abschn. 9.2.1 simulierten Daten mit zeitlich konstanten Prädiktoren und höchstens einmal auftretenden Ereignissen angepasst werden.

```
> library(survival) # für survreg()
> fitWeib <- survreg(Surv(obsT, status) ~ X + IV, dist="weibull",
+ data=dfSurv)
```

Wald-Tests der Parameter sowie einen Likelihood-Quotienten-Test des Gesamtmodells erhält man mit `summary((survreg-Objekt))`.

```
> summary(fitWeib) # Parameter- und Modelltests
Call:
survreg(formula = Surv(obsT, status) ~ X + IV, data = dfSurv,
       dist = "weibull")
      Value Std. Error      z      p
(Intercept) 3.423     0.1703 20.10 7.45e-90
X           -0.632     0.1024 -6.17 6.65e-10
IVB          0.504     0.2449  2.06 3.94e-02
IVC          -0.778     0.2327 -3.34 8.29e-04
Log(scale)   0.216     0.0608  3.56 3.73e-04
Scale= 1.24
```

```
Weibull distribution
Loglik(model)= -695.9   Loglik(intercept only)= -726.9
      Chisq= 62.03 on 3 degrees of freedom, p= 2.2e-13
Number of Newton-Raphson Iterations: 5
n= 180
```

Die Ausgabe ist weitgehend analog zu jener von `summary(<coxph-Objekt>)` (vgl. Abschn. 9.4), wobei in der Spalte `Value` die geschätzten AFT-Parameter $\hat{\gamma}_j = -\hat{\beta}_j \cdot \hat{a}$ genannt werden.⁷ Die Schätzung \hat{a} des Formparameters der Weibull-Verteilung ist unter `Scale` aufgeführt. Der zugehörige Wald-Test mit der $H_0: \ln a = 0$ steht in der Zeile `Log(scale)`. Eine Alternative hierzu ist der Likelihood-Quotienten-Test des eingeschränkten Modells `(fitR)` mit Exponentialverteilung ($a = 1$, also $\ln a = 0$) gegen das umfassendere Modell `(fitU)` mit Weibull-Verteilung mittels `anova(<fitR>, <fitU>)` (vgl. Abschn. 8.1.5).

```
# eingeschränktes Modell mit a=1 -> Exponentialverteilung
> fitExp <- survreg(Surv(obsT, status) ~ X + IV, dist="exponential",
+                      data=dfSurv)

> anova(fitExp, fitWeib)                                # LQ-Modelltest
  Terms Resid. Df   -2*LL Test Df Deviance     Pr(>Chi)
1 X + IV       176 1405.946      NA      NA      NA
2 X + IV       175 1391.839  = 1 14.10752 0.0001726517
```

Da der hier im Modell berücksichtigte Faktor `IV` mit mehreren Parametern γ_j assoziiert ist, muss seine Signifikanz insgesamt über einen Modellvergleich getestet werden. Dazu dient wie beim Vergleich der Modelle mit Exponential- und Weibull-Verteilung ein Likelihood-Quotienten-Test zweier hierarchischer Modelle.

```
# eingeschränktes Modell ohne Faktor IV
> fitR <- survreg(Surv(obsT, status) ~ X, dist="weibull", data=dfSurv)
> anova(fitR, fitWeib)                                # LQ-Modelltest für Faktor IV
  Terms Resid. Df   -2*LL Test Df Deviance     Pr(>Chi)
1      X       177 1418.773      NA      NA      NA
2 X + IV       175 1391.839  +IV  2 26.93433 1.416721e-06
```

9.5.4 Survival-Funktion schätzen

Die geschätzte Verteilungsfunktion $\hat{F}(t)$ für ein mit `survreg()` angepasstes Modell ermittelt `predict()` (vgl. Abschn. 6.4). Die meist stattdessen betrachtete geschätzte Survival-Funktion ergibt sich daraus als $\hat{S}(t) = 1 - \hat{F}(t)$.

```
> predict(<survreg-Objekt>, newdata=<Datensatz>, type="quantile",
+          p=(Quantile), se=TRUE)
```

⁷Für die $\hat{\beta}_j$ ergibt sich 0.51 (`X`), -0.41 (`IVB`) und 0.63 (`IVC`) – also Schätzungen, die hier denen des Cox PH-Modells sehr ähnlich sind (s. S. 335).

Als erstes Argument ist ein `survreg`-Objekt zu übergeben. Das Argument `newdata` erwartet einen Datensatz, der neue Daten für Variablen mit denselben Namen, und bei Faktoren zusätzlich denselben Stufen wie jene der ursprünglichen Prädiktoren im `survreg`-Objekt enthält. Mit dem Argument `type="quantile"` liefert `predict()` für jede Zeile in `newdata` für das Quantil $p \in (0, 1)$ den Wert $\hat{F}^{-1}(p)$. Dies ist die Überlebenszeit t_p , für die bei den in `newdata` gegebenen Gruppenzugehörigkeiten und Prädiktorwerten $\hat{F}(t_p) = p$ gilt. Dafür ist an `p` ein Vektor mit Quantilen zu übergeben, deren zugehörige Werte von $\hat{F}(t)$ gewünscht werden. Den geschätzten Median der Überlebenszeit erfährt man etwa mit `p=0.5`, während für einen durchgehenden Funktionsgraphen von $\hat{F}(t)$ bzw. $\hat{S}(t)$ eine fein abgestufte Sequenz im Bereich $(0, 1)$ angegeben werden muss. Setzt man `se=TRUE`, erhält man zusätzlich noch die geschätzte Streuung für t_p .

Mit `se=TRUE` ist das zurückgegebene Objekt eine Liste mit den Werten von $\hat{F}^{-1}(p)$ in der Komponente `fit` und den geschätzten Streuungen in der Komponente `se.fit`. Umfasst `newdata` mehrere Zeilen, sind `fit` und `se.fit` Matrizen mit einer Zeile pro Beobachtungsobjekt und einer Spalte pro Quantil.⁸

```
# Datensatz: 2 Männer mit Prädiktor X=0 in Gruppe A bzw. in C
> dfNew <- data.frame(sex=factor(c("m", "m"), levels=levels(dfSurv$sex)),
+                         X=c(0, 0),
+                         IV=factor(c("A", "C"), levels=levels(dfSurv$IV)))

# geschätzte Werte von F^(-1)
> percs <- (1:99)/100                                # Perzentile = Quantile
> FWeib <- predict(fitWeib, newdata=dfNew, type="quantile",
+                   p=percs, se=TRUE)

# stelle geschätzte Survival-Funktion S(t) statt F(t) dar -> 1-percs
# zunächst für Beobachtungsobjekt 1
> matplot(cbind(FWeib$fit[1, ],
+                 FWeib$fit[1, ] - 2*FWeib$se.fit[1, ],
+                 FWeib$fit[1, ] + 2*FWeib$se.fit[1, ]),
+            1-percs,
+            type="l", main=expression(paste("Weibull-Fit ", hat(S)(t),
+                                         " mit SE")), xlab="t", ylab="Survival", lty=c(1, 2, 2),
+            lwd=2, col="blue")

# für Beobachtungsobjekt 2
> matlines(cbind(FWeib$fit[2, ],
+                  FWeib$fit[2, ] - 2*FWeib$se.fit[2, ],
+                  FWeib$fit[2, ] + 2*FWeib$se.fit[2, ]),
+                1-percs,
+                col="red", lwd=2)

> legend(x="topright", lwd=2, lty=c(1, 2, 1, 2),
+          col=c("blue", "blue", "red", "red"),
+          legend=c("sex=m, X=0, IV=A", "+- 2*SE",
+                  "sex=m, X=0, IV=C", "+- 2*SE"))
```

⁸ Abschnitt 9.4.2 demonstriert die analoge Verwendung von `newdata` in `survfit()`.

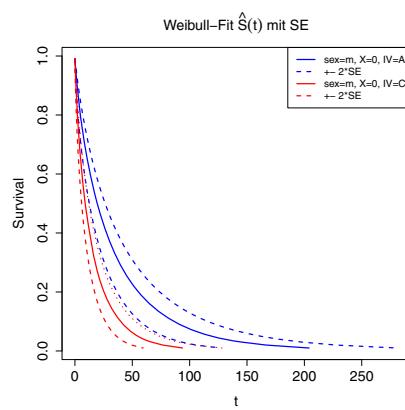


Abbildung 9.7: Schätzung der Survival-Funktion $\hat{S}(t)$ aus Weibull-Modell für zwei männliche Personen mit Prädiktorwert $X = 0$ aus Gruppe A bzw. C

Kapitel 10

Klassische nonparametrische Methoden

Wenn inferenzstatistische Tests zur Datenauswertung herangezogen werden sollen, aber davon ausgegangen werden muss, dass strenge Anforderungen an die Art und Qualität der erhobenen Daten nicht erfüllt sind, kommen viele konventionelle Verfahren womöglich nicht in Betracht. Für solche Situationen hält der Bereich der nonparametrischen Statistik Methoden bereit, deren Voraussetzungen gewöhnlich weniger restriktiv sind und die auch bei kleinen Stichproben zur Auswertung in Frage kommen (Bortz, Lienert & Boehnke, 2010; Büning & Trenkler, 1994). Auch für (gemeinsame) Häufigkeiten kategorialer Variablen und ordinale Daten¹ sind viele der folgenden Methoden geeignet und ergänzen damit die in Kap. 8 vorgestellten Modelle.

Einige der klassischen nonparametrischen Methoden approximieren die unbekannte oder nicht praktisch berechenbare Verteilung der verwendeten Teststatistik durch eine mit wachsender Stichprobengröße asymptotisch gültige Verteilung, die analytisch bestimmbar ist. Einen anderen Weg gehen die in Kap. 11 vorgestellten Resampling-Verfahren (bootstrap und Permutationstests), die die Verteilung der Teststatistik allein auf Basis der gezogenen Stichprobe und vieler zufälliger neuer Versionen von ihr schätzen.

10.1 Anpassungstests

Viele Tests setzen voraus, dass die Verteilung der AV in den untersuchten Bedingungen bekannt ist und bestimmte Voraussetzungen erfüllt – oft muss es sich etwa um eine Normalverteilung handeln. Ob die erhobenen Werte in einer bestimmten Stichprobe mit einer solchen Annahme verträglich sind, kann mit verschiedenen Anpassungstests geprüft werden.

Häufig werden Anpassungstests mit der H_0 durchgeführt, dass eine bestimmte Verteilung vorliegt und dieses Vorliegen auch den gewünschten Zustand beschreibt. Da hier die H_1 gegen die fälschliche Nicht-Annahme abzusichern ist, muss der β -Fehler kontrolliert, also eine gewisse power gesichert werden. Mitunter wird dafür das α -Niveau höher als üblich gewählt (in der Größenordnung von 0.2), auch wenn sich der β -Fehler so nicht exakt begrenzen lässt.

¹ Auf Rangdaten basierende Tests machen häufig die Voraussetzung, dass die Ränge eindeutig bestimmbar sind, also keine gleichen Werte (*Bindungen, ties*) auftauchen. Für den Fall, dass dennoch Bindungen vorhanden sind, existieren unterschiedliche Strategien, wobei die von R-Funktionen gewählte häufig in der zugehörigen Hilfe erwähnt wird.

10.1.1 Binomialtest

Der Binomialtest ist auf Daten von Variablen anzuwenden, die nur zwei Ausprägungen annehmen können. Eine Ausprägung soll dabei als *Treffer* bzw. *Erfolg* bezeichnet werden. Der Test prüft, ob die empirische Auftretenshäufigkeit eines Treffers in einer Stichprobe verträglich mit der H_0 einer bestimmten Trefferwahrscheinlichkeit p_0 ist.

```
> binom.test(x=<Erfolge>, n=<Stichprobengröße>, p=0.5, conf.level=0.95,
+             alternative=c("two.sided", "less", "greater"))
```

Unter `x` ist die beobachtete Anzahl der Erfolge anzugeben, `n` steht für die Stichprobengröße. Alternativ zur Angabe von `x` und `n` kann als erstes Argument ein Vektor mit zwei Elementen übergeben werden, dessen Einträge die Anzahl der Erfolge und Misserfolge sind – etwa das Ergebnis einer Häufigkeitsauszählung mit `xtabs()`. Unter `p` ist p_0 einzutragen. Das Argument `alternative` bestimmt, ob zweiseitig ("two.sided"), links- ("less") oder rechtsseitig ("greater") getestet wird. Die Aussage bezieht sich dabei auf die Reihenfolge p_1 "less" bzw. "greater" p_0 , mit p_1 als Trefferwahrscheinlichkeit unter H_1 . Mit dem Argument `conf.level` wird die Breite des Konfidenzintervalls für p festgelegt.²

Als Beispiel sollen aus einer (unendlich großen) Urne zufällig Lose gezogen werden, wobei $p_0 = 0.25$ ist. Es sei nach der Wahrscheinlichkeit gefragt, bei mindestens 5 von insgesamt 7 Ziehungen einen Gewinn zu ziehen.

```
> draws <- 7                      # Stichprobenumfang
> hits  <- 5                      # Anzahl Treffer
> pH0   <- 0.25                   # Trefferwahrscheinlichkeit unter H0
> binom.test(hits, draws, p=pH0, alternative="greater", conf.level=0.95)
Exact binomial test
data: hits and draws
number of successes = 5, number of trials = 7, p-value = 0.01288
alternative hypothesis: true probability of success is greater than 0.25
95 percent confidence interval:
0.3412614 1.0000000
sample estimates:
probability of success
0.7142857
```

Das Ergebnis enthält neben einer Zusammenfassung der eingegebenen Daten (`number of successes` und `number of trials`, `probability of success`) den p -Wert (`p-value`). Schließlich wird je nach Fragestellung das zwei-, links- oder rechtsseitige Konfidenzintervall für die Trefferwahrscheinlichkeit in der gewünschten Breite genannt. Der ausgegebene p -Wert kann manuell mit Hilfe der Verteilungsfunktion der Binomialverteilung verifiziert werden (vgl. Abschn. 5.3.2, Fußnote 7).

```
> (pVal <- pbinom(hits-1, draws, pH0, lower.tail=FALSE))
[1] 0.01287842
```

²Das Intervall ist jenes nach Clopper-Pearson. Für die Berechnung u. a. nach Wilson, Agresti-Coull und Jeffreys vgl. `BinomCI()` aus dem Paket `DescTools`.

Für den zweiseitigen Binomialtest existieren verschiedene Definitionen des p -Wertes als Wahrscheinlichkeit unter H_0 , mindestens so extreme Trefferzahlen wie die beobachtete zu erhalten:

1. R summiert die Wahrscheinlichkeiten unter Gültigkeit der H_0 für alle Ereignisse mit höchstens der Wahrscheinlichkeit des eingetretenen Ereignisses. So ist etwa der von `binom.test(10, 20, p=0.25, alternative="two.sided")` ausgegebene p -Wert gleich `sum(dbinom(10:20, 20, 0.25)) + sum(dbinom(0, 20, 0.25))`. Die „Extremheit“ eines Ereignisses wird hier also i. S. seiner Wahrscheinlichkeit verstanden.
2. Zwei andere Definitionen beziehen sich auf die separaten einseitigen Tests: Für sie muss im Beispiel zum einen die Wahrscheinlichkeit von 5 oder mehr Gewinnen unter H_0 berechnet werden, zum anderen die Wahrscheinlichkeit von 2 (also 7 – 5) Treffern oder weniger. Der zweiseitige p -Wert kann nun als das Doppelte des kleineren der p -Werte der einseitigen Tests definiert werden.
3. Als weitere Möglichkeit können die p -Werte für jede der beiden einseitigen Fragestellungen addiert werden.

Wenn unter H_0 mit $p = 0.5$ eine symmetrische Binomialverteilung vorliegt, führen die genannten Definitionen zum selben Ergebnis. Nach einer gängigen Konvention ist beim zweiseitigen Test auf jeder Seite $\frac{\alpha}{2}$ einzuhalten, auch wenn bei Überschreitung von $\frac{\alpha}{2}$ auf einer Seite dennoch insgesamt $p < \alpha$ wäre.

```
# rechtsseitiger Test
> resG <- binom.test(hits, draws, p=pH0, alternative="greater")
> resG$p.value                                # p-Wert
[1] 0.01287842

# linksseitiger Test
> resL <- binom.test(draws-hits, draws, p=pH0, alternative="less")
> resL$p.value                                # p-Wert
[1] 0.7564087

# Gesamt-p-Werte
> 2 * min(resG$p.value, resL$p.value)          # Variante 2
[1] 0.02575684
```

Hypothesen über die jeweiligen Trefferwahrscheinlichkeiten einer dichotomen Variable in zwei unabhängigen Stichproben lassen sich mit Fishers exaktem Test prüfen (vgl. Abschn. 10.2.5), in zwei abhängigen Stichproben mit dem McNemar-Test (vgl. Abschn. 10.5.10), in mehr als zwei unabhängigen Stichproben mit einem χ^2 -Test (vgl. Abschn. 10.2.3) und in mehr als zwei abhängigen Stichproben mit Cochran's Q-Test (vgl. Abschn. 10.5.8).

10.1.2 Test auf Zufälligkeit (Runs-Test)

Eine zufällige Reihenfolge von N Ausprägungen einer dichotomen Variable sollte sich dann ergeben, wenn die Erfolgswahrscheinlichkeit für jede Realisierung konstant p beträgt und die Realisierungen unabhängig sind. Mit dem Test auf Zufälligkeit kann geprüft werden, ob eine empirische Datenreihe mit der Annahme von Zufälligkeit verträglich ist. Teststatistik R ist die

Anzahl der Iterationen (*runs*, vgl. Abschn. 2.10.2), wobei eine sehr geringe als auch eine sehr hohe Anzahl gegen die H_0 spricht.

Eine spezielle Funktion für den Test auf Zufälligkeit ist nicht im Basisumfang von R enthalten,³ eine manuelle Durchführung jedoch möglich. Als Beispiel diene jenes aus Bortz et al. (2010, p. 548 ff.): Bei einer Warteschlange aus 8 Personen (5 Frauen, f und 3 Männer, m) wird das Geschlecht des an jeder Position Wartenden erhoben.

```
> queue <- c("f", "m", "m", "f", "m", "f", "f", "f")           # Daten
> Nj      <- xtabs(~ queue)                                # Gruppengrößen
> (runs <- rle(queue))                                    # Iterationen
Run Length Encoding
lengths: int [1:5] 1 2 1 1 3
values: chr [1:5] "f" "m" "f" "m" "f"

> (rr <- length(runs$lengths))                            # Gesamtzahl Iterationen
[1] 5

> (rr1 <- xtabs(~ runs$values)[1])                      # Iterationen Gruppe 1
f
3

> (rr2 <- xtabs(~ runs$values)[2])                      # Iterationen Gruppe 2
m
2
```

Für den *p*-Wert des beobachteten Wertes von *R* sind die Punktwahrscheinlichkeiten für alle Fälle aufzuaddieren, die dieses *R* oder extremere Werte ergeben. Die Berechnung der Punktwahrscheinlichkeiten geschieht hier mit einer eigens erstellten Funktion (vgl. Abschn. 15.2). Für die Ermittlung des *p*-Wertes ist zu beachten, dass ein ungerades *R* generell auf zwei Arten zustande kommen kann: Entweder ist die Anzahl der Iterationen der ersten Gruppe um 1 größer als die der zweiten, oder umgekehrt. Hier sind die Fälle für $R = 5, 6, 7$ zu berücksichtigen – der Obergrenze möglicher Iterationen. Der Fall $R = 7$ kann sich hier nur auf eine Weise ergeben, da nur 3 Männer vorhanden sind, so dass insgesamt 4 Punktwahrscheinlichkeiten zu addieren sind.

```
# Funktion, um Punktwahrscheinlichkeit für Anzahl von Iterationen der
# Gruppe 1 (r1), Gruppe 2 (r2), mit Gruppengrößen n1 und n2 zu berechnen
> getP <- function(r1, r2, n1, n2) {
+   # Punktwahrscheinlichkeit für r1+r2 ungerade
+   p <- (choose(n1-1, r1-1) * choose(n2-1, r2-1)) / choose(n1+n2, n1)
+
+   # Punktwahrscheinlichkeit für r1+r2 gerade: das Doppelte von ungerade
+   ifelse(((r1+r2) %% 2) == 0, 2*p, p)
+ }
```



```
> n1    <- Nj[1]                                         # Größe Gruppe 1
```

³Vergleiche hierfür `RunsTest()` aus dem `DescTools` Paket.

```

> n2    <- Nj[2]                                # Größe Gruppe 2
> N     <- sum(Nj)                             # Gesamt-N
> rMin <- 2                                     # Untergrenze für Anzahl der Iterationen

# Obergrenze Anzahl Iterationen, Fallunterscheidung: n1 == n2?
> (rMax <- ifelse(n1 == n2, N, 2*min(n1, n2) + 1))
[1] 7

# addiere Punktwahrscheinlichkeiten für R=beobachtet und größer
> p3.2   <- getP(3, 2, n1, n2)                  # r1=3, r2=2 -> R=5
> p2.3   <- getP(2, 3, n1, n2)                  # r1=2, r2=3 -> R=5
> p3.3   <- getP(3, 3, n1, n2)                  # r1=3, r2=3 -> R=6
> p4.3   <- getP(4, 3, n1, n2)                  # r1=4, r2=3 -> R=7
> (pGrEq <- p3.2 + p2.3 + p3.3 + p4.3)       # p-Wert einseitig
0.5714286

# Punktwahrscheinlichkeit aller anderen Fälle
> p2.2   <- getP(2, 2, n1, n2)                  # r1=2, r2=2 -> R=4
> p1.2   <- getP(1, 2, n1, n2)                  # r1=1, r2=2 -> R=3
> p2.1   <- getP(2, 1, n1, n2)                  # r1=2, r2=1 -> R=3
> p1.1   <- getP(1, 1, n1, n2)                  # r1=1, r2=1 -> R=2
> (pLess <- p2.2 + p1.2 + p2.1 + p1.1)
0.4285714

> pGrEq + pLess                               # Kontrolle: Summe beider Wkt. = 1
1

```

Aus R lässt sich eine mit wachsender Stichprobengröße asymptotisch standardnormalverteilte Teststatistik berechnen, deren Verwendung ab einer Anzahl von ca. 30 Beobachtungen nur zu geringen Fehlern führen sollte.

```

> muR   <- 1 + ((2*n1*n2) / N)                # Erwartungswert von R
> varR  <- (2*n1*n2*(2*n1*n2 - N)) / (N^2 * (N-1))      # Varianz von R
> rZ    <- (rr-muR) / sqrt(varR)               # z-Transformierte von R
> (pVal <- pnorm(rZ, lower.tail=FALSE))        # p-Wert einseitig
0.4184066

```

In Form des Wald-Wolfowitz-Tests lässt sich auch prüfen, ob zwei Stichproben aus derselben Grundgesamtheit stammen bzw. ob die zu ihnen gehörenden Variablen dieselbe Verteilung besitzen. Dazu werden die Daten beider Stichproben gemeinsam ihrer Größe nach geordnet. Anschließend ist jeder Wert durch die Angabe zu ersetzen, aus welcher Stichprobe er stammt und der Test wie jener auf Zufälligkeit durchzuführen.

10.1.3 Kolmogorov-Smirnov-Anpassungstest

Der Kolmogorov-Smirnov-Test auf eine feste Verteilung ist als exakter Test auch bei kleinen Stichproben anwendbar und vergleicht die kumulierten relativen Häufigkeiten von Daten einer

stetigen Variable mit einer frei wählbaren Verteilungsfunktion – etwa der einer bestimmten Normalverteilung. Gegen die H_0 , dass die Verteilungsfunktion gleich der angegebenen ist, kann eine ungerichtete wie gerichtete H_1 getestet werden. Der Test lässt sich durch eine visuell-explorative Analyse mittels eines Quantil-Quantil-Diagramms ergänzen (vgl. Abschn. 14.6.5).

```
> ks.test(x=<Vektor>, y="(Name der Verteilungsfunktion)", ...,
+           alternative=c("two.sided", "less", "greater"))
```

Unter x ist der Datenvektor einzugeben und unter y die Verteilungsfunktion der Variable unter H_0 . Um durch Komma getrennte Argumente an diese Verteilungsfunktion zu ihrer genauen Spezifikation übergeben zu können, dienen die ... Auslassungspunkte. Mit `alternative` wird die H_1 definiert, wobei sich `"less"` und `"greater"` darauf beziehen, ob y stochastisch kleiner oder größer als x ist.⁴

Im Beispiel soll zum Test die Verteilungsfunktion der Normalverteilung mit Erwartungswert $\mu = 1$ und Streuung $\sigma = 2$ herangezogen werden.

```
> DV <- rnorm(8, mean=1.5, sd=3)                      # Daten
> ks.test(DV, "pnorm", mean=1, sd=2, alternative="two.sided")
One-sample Kolmogorov-Smirnov test
data: DV
D = 0.2326, p-value = 0.6981
alternative hypothesis: two-sided
```

Die Ausgabe umfasst den empirischen Wert der zweiseitigen Teststatistik (D) sowie den zugehörigen p -Wert (`p-value`). Im folgenden wird die Teststatistik sowohl für den ungerichteten wie für beide gerichteten Tests manuell berechnet, zudem sollen die hierfür relevanten Differenzen zwischen empirischer und angenommener Verteilung grafisch veranschaulicht werden (Abb. 10.1).

```
> Fn      <- ecdf(DV)          # Funktion für kumulierte rel. Häufigkeiten
> sortDV <- sort(DV)          # sortierte Daten
> emp     <- Fn(sortDV)       # kumulierte relative Häufigkeiten

# theoretische Werte: Verteilungsfkt. der Normalverteilung N(mu=1, sigma=2)
> theo   <- pnorm(sortDV, mean=1, sd=2)
> diff1 <- emp - theo         # direkte Differenzen
> diff2 <- c(0, emp[-length(emp)]) - theo        # verschobene Differ.

# Teststatistik für zweiseitigen Test: maximale absolute Abweichung
> (DtwoS <- max(abs(c(diff1, diff2))))
[1] 0.2325919

# Teststatistik für "less": Betrag der stärksten Abweichung nach unten
> (Dless <- abs(min(c(diff1, diff2))))
```

⁴Bei zwei Zufallsvariablen X und Y ist Y dann stochastisch größer als X , wenn die Verteilungsfunktion von Y an jeder Stelle unter der von X liegt. Besitzen X und Y etwa Verteilungen derselben Form, ist dies der Fall, wenn die Dichte- bzw. Wahrscheinlichkeitsfunktion von Y eine nach rechts verschobene Version der von X darstellt.

```
[1] 0.2325919

# Teststatistik für "greater": Betrag der stärksten Abweichung nach oben
> (Dgreat <- abs(max(c(diff1, diff2))))
[1] 0.09828234

# Kontrolle über Ausgabe von ks.test()
> ks.test(DV, "pnorm", mean=1, sd=2, alternative="less")$statistic
D^-
0.2325919

> ks.test(DV, "pnorm", mean=1, sd=2, alternative="greater")$statistic
D^+
0.09828234

# grafische Darstellung der direkten und verschobenen Differenzen
> plot(Fn, main="Kolmogorov-Smirnov-Anpassungstest", xlab=NA)
> curve(pnorm(x, mean=1, sd=2), n=200, add=TRUE)      # Verteilungsfunktion

# direkte Abweichungen
> matlines(rbind(sortDV, sortDV), rbind(emp, theo),
+           col=rgb(0, 0, 1, 0.7), lty=1, lwd=2)

# verschobene Abweichungen
> matlines(rbind(sortDV, sortDV), rbind(c(0, emp[1:(length(emp)-1)]),
+           theo), col=rgb(1, 0, 0, 0.5), lty=1, lwd=2)

> legend(x="topleft", legend=c("direkte Differenzen",
+           "verschobene Differenzen"), col=c("blue", "red"), lwd=2)
```

Der beschriebene Kolmogorov-Smirnov-Test setzt voraus, dass die theoretische Verteilungsfunktion vollständig festgelegt ist, also keine zusammengesetzte H_0 vorliegt, die nur die Verteilungsklasse benennt. Für einen korrekten Test dürfen die Parameter nicht auf Basis der Daten in `x` geschätzt werden. Für den Test auf Normalverteilung stellt das Paket `DescTools` mit `LillieTest()` die Abwandlung mit Lilliefors-Schranken sowie mit `AndersonDarlingTest()` den Anderson-Darling-Test zur Verfügung, die diese Voraussetzungen nicht machen.

Der über `shapiro.test()` aufzurufende Shapiro-Wilk-Test auf Normalverteilung ist eine dem Lilliefors-Test oft vorgezogene Alternative.

Der Kolmogorov-Smirnov-Test lässt sich auch verwenden, um die Daten einer stetigen Variable aus zwei Stichproben daraufhin zu prüfen, ob sie mit der H_0 verträglich sind, dass die Variable in beiden Bedingungen dieselbe Verteilung besitzt (vgl. Abschn. 10.5.1).

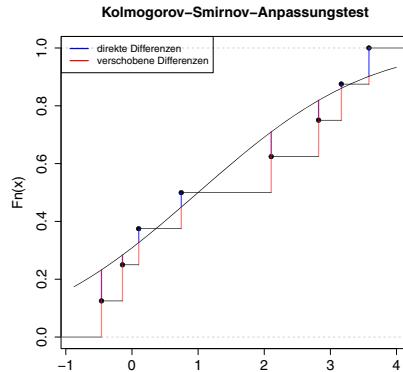


Abbildung 10.1: Kolmogorov-Smirnov-Anpassungstest: Abweichungen zwischen kumulierten relativen Häufigkeiten der Daten und der Verteilungsfunktion der Normalverteilung $\mathcal{N}(\mu = 1, \sigma = 2)$

10.1.4 χ^2 -Test auf eine feste Verteilung

Der χ^2 -Test auf eine feste Verteilung prüft Daten einer kategorialen Variable daraufhin, ob die empirischen Auftretenshäufigkeiten der einzelnen Kategorien verträglich mit einer theoretischen Verteilung unter H_0 sind, die die Wahrscheinlichkeit jeder Kategorie angibt.⁵ Als ungerichtete H_1 ergibt sich, dass die tatsächliche Verteilung nicht gleich der unter H_0 genannten ist.

```
> chisq.test(x=<Häufigkeiten>, p=<Wahrscheinlichkeiten>,
+             simulate.p.value=FALSE)
```

Unter x ist der Vektor anzugeben, der die empirischen absoluten Auftretenshäufigkeiten der Kategorien beinhaltet. Dies kann z. B. eine Häufigkeitstabelle als Ergebnis von `xtabs()` sein. Unter p ist ein Vektor einzutragen, der die Wahrscheinlichkeiten für das Auftreten der Kategorien unter H_0 enthält und dieselbe Länge wie x haben muss. Treten erwartete Häufigkeiten von Kategorien (das Produkt der Klassenwahrscheinlichkeiten mit der Stichprobengröße) < 5 auf, sollte das Argument `simulate.p.value` auf `TRUE` gesetzt werden. Andernfalls gibt R in einer solchen Situation die Warnung aus, dass die χ^2 -Approximation noch unzulänglich sein kann.

Als Beispiel soll ein empirischer Würfel daraufhin getestet werden, ob er fair ist. Die Auftretenswahrscheinlichkeit jeder Augenzahl unter H_0 beträgt $\frac{1}{6}$.

```
> nRolls <- 50                                # Anzahl Ziehungen
> nCateg <- 6                                  # Anzahl Kategorien
> pH0    <- rep(1/nCateg, nCateg)              # Verteilung unter H0
> myData <- sample(1:nCateg, nRolls, replace=TRUE) # simulierte Daten
> (tab   <- xtabs(~ myData))                  # Häufigkeiten
```

⁵Dieser Test ist auch bei quantitativen Variablen durchführbar, wobei zunächst eine Einteilung der Werte in disjunkte Klassen vorzunehmen ist. Die getestete H_0 ist dann in dem Sinne schwächer, dass alle Verteilungen äquivalent sind, die zu gleichen Klassenwahrscheinlichkeiten führen.

```
myData
1 2 3 4 5 6
8 8 4 14 3 13

> chisq.test(tab, p=pH0)
Chi-squared test for given probabilities
data: tab
X-squared = 12.16, df = 5, p-value = 0.03266
```

Die Ausgabe umfasst den Wert der mit wachsender Stichprobengröße asymptotisch χ^2 -verteilten Teststatistik (**X-squared**) und ihre Freiheitsgrade (**df**, nur wenn `simulate.p.value` gleich `FALSE` ist) samt zugehörigem *p*-Wert (**p-value**). Das Ergebnis lässt sich manuell prüfen:

```
> expected <- pH0 * nRolls      # erwartete Häufigkeiten unter H0
> (statChisq <- sum((tab-expected)^2 / expected))    # Teststatistik
[1] 12.16

> (pVal <- pchisq(statChisq, nCateg-1, lower.tail=FALSE))    # p-Wert
[1] 0.03265998
```

10.1.5 χ^2 -Test auf eine Verteilungsklasse

Als Spezialfall des χ^2 -Tests auf Gleichheit von Verteilungen (vgl. Abschn. 10.2.2) kann die Verträglichkeit der Daten mit der H_0 getestet werden, dass die zugehörige theoretische Verteilung etwa aus der Familie der Normalverteilungen stammt – andere Verteilungsfamilien lassen sich analog testen. Die H_1 ist ungerichtet, der Test asymptotisch korrekt bei wachsender Stichprobengröße und kann durch eine visuell-explorative Begutachtung eines Quantil-Quantil-Diagramms ergänzt werden (vgl. Abschn. 14.6.5).

Die Daten sind zunächst in disjunkte Klassen einzuteilen, deren empirische Auftretenshäufigkeiten mit den unter H_0 erwarteten verglichen werden.⁶ Die Parameter der konkreten Normalverteilung unter H_0 folgen entweder aus theoretischen Erwägungen oder werden auf Basis der Daten geschätzt, häufig anhand des Mittelwertes und der korrigierten Stichprobenstreitung. Durch eine solche Schätzung beider Parameter aus den Daten reduziert sich die Anzahl der Freiheitsgrade beim Test um 2.⁷

Das Paket `DescTools` enthält mit `PearsonTest()` eine Funktion für den χ^2 -Test auf Normalverteiltheit, über die sowohl die Klasseneinteilung wie auch die Korrektur der Freiheitsgrade per Argument festgelegt werden können.

```
> PearsonTest(x=<Vektor>, n.classes=<Anzahl>, adjust=TRUE)
```

Der Datenvektor wird der Funktion als Argument `x` übergeben, `n.classes` legt die Zahl der zu bildenden Klassen fest. Die Funktion wählt die genaue Lage der Klassengrenzen so, dass alle

⁶Die Klassenbildung führt dazu, dass statt der Verträglichkeit mit einer bestimmten Verteilung die schwächere H_0 einer Verträglichkeit mit allen Verteilungen getestet wird, die zu denselben erwarteten Häufigkeiten führen.

⁷Für eine korrekte Testkonstruktion wäre eigentlich eine feste Klasseneinteilung mit gruppierter Maximum-Likelihood- oder Minimum- χ^2 -Schätzung von μ und σ notwendig.

Klassen unter H_0 dieselbe Wahrscheinlichkeit besitzen. Über `adjust` wird angegeben, ob die Zahl der Freiheitsgrade in Folge einer Schätzung der Parameter der Normalverteilung unter H_0 aus den Daten um 2 nach unten zu korrigieren ist.

```
> DV <- rnorm(100, mean=100, sd=15) # simulierte Daten
> nCls <- 6 # Anzahl Klassen
> library(DescTools) # für PearsonTest()
> PearsonTest(DV, n.classes=nCls, adjust=TRUE)
Pearson chi-square normality test
data: DV
P = 3.08, p-value = 0.3795
```

Die Ausgabe nennt den empirischen χ^2 -Wert unter P zusammen mit dem zugehörigen p -Wert. Für einen manuellen Test sollen ebenfalls gleichwahrscheinliche Intervalle als Basis dienen. Deren innere Grenzen liefert `qnorm()`, wobei Erwartungswert und Streuung aus den Daten geschätzt werden.

```
# innere Intervallgrenzen für gleichwahrscheinliche Intervalle
> limits <- qnorm(seq(1/nCls, (nCls-1)/nCls, length.out=nCls-1),
+                  mean(DV), sd(DV))

> DVCut <- cut(DV, c(-Inf, limits, Inf)) # + äußere Int.grenzen
> (intFreq <- xtabs(~ DVCut)) # beobachtete Häufigk.
DVCut
(-Inf,87.3] (87.3,95.1] (95.1,101] (101,108] (108,115] (115, Inf]
           19          14          14          16          22          15
```

Beim Vergleich von beobachteten und erwarteten Klassenhäufigkeiten ist zu berücksichtigen, dass der von `chisq.test()` ausgegebene p -Wert nicht die Reduktion der Freiheitsgrade widerspiegelt und deshalb für diesen Test nicht der richtige ist. Ist `⟨Objekt⟩` das Ergebnis von `chisq.test()`, muss die Bestimmung des korrekten p -Wertes manuell mit dem empirischen Wert der Teststatistik, den richtigen Freiheitsgraden und der Verteilungsfunktion `pchisq(⟨Quantil⟩, ⟨Freiheitsgrade⟩)` erfolgen.

```
# Test für gleiche Klassenwahrscheinlichkeiten, falsche Freiheitsgrade
> (resChisq <- chisq.test(intFreq, p=rep(1/nCls, nCls)))
Chi-squared test for given probabilities
data: intFreq
X-squared = 3.08, df = 5, p-value = 0.6877

# Test mit den richtigen Freiheitsgraden
> statChisq <- resChisq$statistic # Teststatistik chi^2
> realDf <- resChisq$parameter - 2 # korrig. Freiheitsgr.

# korrigierter p-Wert
> (realPval <- pchisq(statChisq, realDf, lower.tail=FALSE))
0.3794545
```

10.2 Analyse von gemeinsamen Häufigkeiten kategorialer Variablen

Inwieweit die gemeinsamen empirischen Auftretenshäufigkeiten der Ausprägungen kategorialer Variablen theoretischen Vorstellungen entsprechen, kann durch folgende Tests geprüft werden. Sie ergänzen die in Kap. 8 vorgestellten Regressionsmodelle für bestimmte Spezialfälle.

10.2.1 χ^2 -Test auf Unabhängigkeit

Beim χ^2 -Test auf Unabhängigkeit wird die empirische Kontingenztafel von zwei an derselben Stichprobe erhobenen kategorialen Variablen daraufhin geprüft, ob sie verträglich mit der H_0 ist, dass beide Variablen unabhängig sind.⁸ Die H_1 ist ungerichtet, der Test asymptotisch korrekt bei wachsender Stichprobengröße.

```
> chisq.test(x, y=NULL, simulate.p.value=FALSE)
```

Unter x kann eine zweidimensionale Kontingenztafel eingegeben werden – etwa als Ergebnis von `xtabs(~ Faktor1 + Faktor2)`, oder als Matrix mit den Häufigkeiten der Stufenkombinationen zweier Variablen. Alternativ kann x ein Objekt der Klasse `factor` mit den Ausprägungen der ersten Variable sein. In diesem Fall muss auch y angegeben werden, das dann ebenfalls ein Objekt der Klasse `factor` derselben Länge wie x mit an denselben Beobachtungsobjekten erhobenen Daten zu sein hat.

Unter H_0 ergeben sich die Zellwahrscheinlichkeiten jeweils als Produkt der zugehörigen Randwahrscheinlichkeiten, die über die relativen Randhäufigkeiten geschätzt werden. Das Argument `simulate.p.value=TRUE` sollte gesetzt werden, wenn erwartete Zellhäufigkeiten < 5 auftreten. Andernfalls gibt R in einer solchen Situation die Warnung aus, dass die χ^2 -Approximation noch unzulänglich sein kann.

Als Beispiel sei eine Stichprobe von Studenten betrachtet, die angeben, ob sie rauchen und wie viele Geschwister sie haben.

```
> N      <- 50                                # Stichprobengröße
> smokes <- factor(sample(c("no", "yes"), N, replace=TRUE))
> siblings <- factor(round(abs(rnorm(N, 1, 0.5))))
> cTab    <- xtabs(~ smokes + siblings)       # Kontingenztafel
> addmargins(cTab)                           # Randsummen
      siblings
smokes 0 1 2 Sum
  no   6 18 1 25
  yes  2 18 5 25
  Sum   8 36 6 50

> chisq.test(cTab)
Pearson's Chi-squared test
data: cTab
```

⁸Dieser Test ist auch bei quantitativen Variablen durchführbar, wobei zunächst eine Einteilung der Werte in disjunkte Klassen vorzunehmen ist. Die getestete H_0 ist dann in dem Sinne schwächer, dass nur die Unabhängigkeit bzgl. der vorgenommenen Klasseneinteilung getestet wird.

```
X-squared = 4.6667, df = 2, p-value = 0.09697
```

Warning message:

```
In chisq.test(cTab) : Chi-squared approximation may be incorrect
```

Das Ergebnis lässt sich manuell kontrollieren:

```
> P <- nlevels(smokes)                                # Anzahl smokes Kategorien
> Q <- nlevels(siblings)                             # Anzahl siblings Kategorien

# erwartete Häufigkeiten -> alle Produkte der Randhäufigkeiten
# relativiert an der Summe von Beobachtungen
> expected <- outer(rowSums(cTab), colSums(cTab)) / sum(cTab)
> (statChisq <- sum((cTab-expected)^2 / expected))    # Teststatistik
[1] 4.666667

> (pVal <- pchisq(statChisq, (P-1)*(Q-1), lower.tail=FALSE)) # p-Wert
[1] 0.09697197
```

Vergleiche Abschn. 8.5 für eine Verallgemeinerung zur Analyse höherdimensionaler Kontingenztafeln absoluter Häufigkeiten mit Hilfe log-linearer Modelle.

10.2.2 χ^2 -Test auf Gleichheit von Verteilungen

Beim χ^2 -Test auf Gleichheit von bedingten Verteilungen werden die empirischen eindimensionalen Häufigkeitsverteilungen einer kategorialen Variable in unterschiedlichen Stichproben daraufhin geprüft, ob sie mit der H_0 verträglich sind, dass ihre Stufen in allen Bedingungen jeweils dieselben Auftretenswahrscheinlichkeiten besitzen.⁹ Die H_1 ist ungerichtet, der Test asymptotisch korrekt bei wachsender Stichprobengröße. Getestet wird wie in Abschn. 10.2.1 mit `chisq.test()` auf Basis der Kontingenztafel gemeinsamer Häufigkeiten.

Als Beispiel soll die politische Orientierung von 100 Studenten aus zwei Studiengängen getestet werden. AV sei die gewählte von fünf Parteien.

```
> voteX <- rep(LETTERS[1:5], c(3, 8, 12, 19, 8))      # Ergebnisse Fach X
> voteY <- rep(LETTERS[1:5], c(8, 17, 16, 7, 2))      # Ergebnisse Fach Y
> vote <- c(voteX, voteY)                               # beide Ergebnisse

# entsprechender Faktor mit Gruppenzugehörigkeiten
> studies <- factor(rep(c("X", "Y"), c(length(voteX), length(voteY))))
> cTab <- xtabs(~ studies + vote)                      # Kontingenztafel
> addmargins(cTab)                                     # Randsummen
   vote
studies A   B   C   D   E   Sum
      X 3   8   12  19   8   50
```

⁹Dieser Test ist auch bei quantitativen Variablen durchführbar, wobei zunächst eine Einteilung der Werte in disjunkte Klassen vorzunehmen ist. Die getestete H_0 ist dann in dem Sinne schwächer, dass nur die Gleichheit der Verteilungen bzgl. der vorgenommenen Klasseneinteilung getestet wird.

Y	8	17	16	7	2	50
Sum	11	25	28	26	10	100

```
# simulate.p.value=TRUE wegen geringen erwarteten Häufigkeiten
> chisq.test(cTab, simulate.p.value=TRUE)
Pearson's Chi-squared test with simulated p-value (based on 2000 replicates)
data: cTab
X-squared = 15.2226, df = NA, p-value = 0.005497
```

10.2.3 χ^2 -Test für mehrere Auftretenswahrscheinlichkeiten

Für eine in mehr als einer Bedingung erhobene dichotome Variable prüft `prop.test()`, ob die Trefferwahrscheinlichkeit in allen Bedingungen dieselbe ist. Es handelt sich also um eine Hypothese zur Gleichheit von Verteilungen einer Variable in mehreren Bedingungen. Der Test ist asymptotisch korrekt bei wachsender Stichprobengröße, die H_1 ist ungerichtet.

```
> prop.test(x=<Erfolge>, n=<Stichprobenumfänge>, p=NULL)
```

Die Argumente `x`, `n` und `p` beziehen sich auf die Anzahl der Treffer, die Stichprobengrößen und die Trefferwahrscheinlichkeiten unter H_0 in den Gruppen. Für sie können Vektoren gleicher Länge mit Einträgen für jede Gruppe angegeben werden. Anstelle von `x` und `n` ist auch eine Matrix mit zwei Spalten möglich, die in jeder Zeile die Zahl der Treffer und Nicht-Treffer für jeweils eine Gruppe enthält. Ohne konkrete Angabe für `p` testet `prop.test()` die Hypothese, dass die Trefferwahrscheinlichkeit in allen Bedingungen dieselbe ist.

Als Beispiel soll die H_0 getestet werden, dass in drei Gruppen, hier gleicher Größe, jeweils dieselbe Trefferwahrscheinlichkeit vorliegt.

```
> total <- c(5000, 5000, 5000)                                # Gruppengrößen
> hits  <- c(585, 610, 539)                                # Anzahl Treffer
> prop.test(hits, total)
3-sample test for equality of proportions without continuity correction
data: hits out of total
X-squared = 5.0745, df = 2, p-value = 0.07908
alternative hypothesis: two.sided
sample estimates:
prop 1  prop 2  prop 3
0.1170  0.1220  0.1078
```

Die Ausgabe berichtet den Wert der χ^2 -Teststatistik (`X-squared`) samt des zugehörigen *p*-Wertes (`p-value`) zusammen mit den Freiheitsgraden (`df`) und schließlich die relativen Erfolgshäufigkeiten in den Gruppen (`sample estimates`). Dasselbe Ergebnis lässt sich auch durch geeignete Anwendung des χ^2 -Tests auf Gleichheit von Verteilungen erzielen. Hierfür müssen zunächst für jede der drei Stichproben auch die Auftretenshäufigkeiten der zweiten Kategorie berechnet und zusammen mit jenen der ersten Kategorie in einer Matrix zusammengestellt werden.

```
> (mat <- cbind(hits, total-hits))      # Matrix mit Treffern und Nieten
[1,] [,1] [,2]
[1,] 585 4415
[2,] 610 4390
[3,] 539 4461

> chisq.test(mat)
Pearson's Chi-squared test
data: mat
X-squared = 5.0745, df = 2, p-value = 0.07908
```

Ergeben sich die Gruppen durch Werte einer ordinalen Variable, ist mit `prop.trend.test()` ebenfalls über eine χ^2 -Teststatistik die spezialisiertere Prüfung möglich, ob die Erfolgswahrscheinlichkeiten der dichotomen Variable einem Trend bzgl. der ordinalen Variable folgen.

10.2.4 Fishers exakter Test auf Unabhängigkeit

Werden zwei dichotome Variablen in einer Stichprobe erhoben, kann mit Fishers exaktem Test die H_0 geprüft werden, dass beide Variablen unabhängig sind.¹⁰ Anders als beim χ^2 -Test zur selben Hypothese sind hier auch gerichtete Alternativhypthesen über den Zusammenhang möglich.

```
> fisher.test(x, y=NULL, conf.level=0.95,
+               alternative=c("two.sided", "less", "greater"))
```

Unter `x` ist entweder die (2×2) -Kontingenztafel zweier dichotomer Variablen oder ein Objekt der Klasse `factor` mit zwei Stufen anzugeben, das die Ausprägungen der ersten dichotomen Variable enthält. In diesem Fall muss auch ein Faktor `y` mit zwei Stufen und derselben Länge wie `x` angegeben werden, der Daten derselben Beobachtungsobjekte beinhaltet. Das Argument `alternative` bestimmt, ob zweiseitig, links- oder rechtsseitig getestet werden soll. Die Richtung der Fragestellung bezieht sich dabei auf die Größe des *odds ratio* (OR) in der theoretischen Kontingenztafel. Linksseitiges Testen bedeutet, dass unter H_1 $OR < 1$ ist (negativer Zusammenhang), rechtsseitiges Testen entsprechend, $OR > 1$ (positiver Zusammenhang). Mit dem Argument `conf.level` wird die Breite des Konfidenzintervalls für das OR festgelegt.

Im Beispiel soll geprüft werden, ob das Ergebnis eines diagnostischen Instruments für eine bestimmte Krankheit wie gewünscht positiv mit dem Vorliegen dieser Krankheit zusammenhängt.

```
# Gruppenzugehörigkeit: 10 Gesunde, 5 Kranke
> disease <- factor(rep(c("no", "yes"), c(10, 5)))
> diagN   <- rep(c("isHealthy", "isIll"), c(8, 2))  # Diagnose für Gesunde
> diagY   <- rep(c("isHealthy", "isIll"), c(1, 4))  # Diagnose für Kranke
> diagT   <- factor(c(diagN, diagY))                 # alle Diagnosen
> contT1  <- xtabs(~ disease + diagT)                # Kontingenztafel
```

¹⁰Die H_0 ist äquivalent zur Hypothese, dass das wahre *odds ratio* der Kontingenztafel beider Variablen gleich 1 ist (vgl. Abschn. 10.2.6). Der Test lässt sich auf Variablen mit mehr als zwei Stufen verallgemeinern, vgl. `?fisher.test`.

```
> addmargins(contT1)                                     # Randsummen
   diagT
disease  isHealthy  isIll  Sum
  no        8       2    10
  yes       1       4     5
  Sum       9       6    15

> fisher.test(contT1, alternative="greater")
Fisher's Exact Test for Count Data
data: contT1
p-value = 0.04695
alternative hypothesis: true odds ratio is greater than 1
95 percent confidence interval:
1.031491 Inf
sample estimates:
odds ratio
12.49706
```

Die Ausgabe enthält neben dem p -Wert (`p-value`) das Konfidenzintervall für das OR in der gewünschten Breite sowie die bedingte Maximum-Likelihood-Schätzung des OR für die gegebenen Randhäufigkeiten (`sample estimates`). Der p -Wert lässt sich manuell nachprüfen: Als Variante eines Permutationstests (vgl. Abschn. 11.2) müssen hierfür die Punktwahrscheinlichkeiten für die vorliegende sowie für i. S. der H_1 extremere Kontingenztafeln bei gleichen Randhäufigkeiten mit der hypergeometrischen Verteilung ermittelt und summiert werden. Im gegebenen Fall besteht nur eine Möglichkeit, die Kontingenztafel extremer zu machen, ohne die Randhäufigkeiten zu ändern.

```
# Punktwahrscheinlichkeit für die gegebene Kontingenztafel
> (p1 <- dhyper(8, 8+2, 1+4, 8+1))
[1] 0.04495504

# Punktwahrscheinlichkeit für extremere Kontingenztafel
> (p2 <- dhyper(9, 9+1, 0+5, 9+0))
[1] 0.001998002

> (pVal <- p1 + p2)      # Summe der Punktwahrscheinlichkeiten -> p-Wert
[1] 0.04695305
```

Die H_0 , dass zwei dichotome Variablen in mehreren, durch eine dritte kategoriale Variable gebildeten Bedingungen jeweils unabhängig sind, prüft der Cochran-Mantel-Hänszel-Test, für den die Funktion `mantelhaen.test()` existiert. Sie ist auch im allgemeineren Fall anwendbar, dass die Unabhängigkeit kategorialer Variablen mit mehr als zwei Stufen zu testen ist.

10.2.5 Fishers exakter Test auf Gleichheit von Verteilungen

Für Daten einer dichotomen Variable aus zwei unabhängigen Stichproben prüft Fishers exakter Test die H_0 , dass die Variablen in beiden Bedingungen identische Erfolgswahrscheinlichkeit

besitzen. Anders als beim χ^2 -Test zur selben Hypothese sind hier auch gerichtete Alternativhypothesen möglich.

Der Aufruf von `fisher.test()` ist identisch zu jenem beim Test auf Unabhängigkeit (vgl. auch Fußnote 10). Unter `x` ist hier entweder die (2×2) -Kontingenztafel einer dichotomen Zufallsvariable und eines Gruppierungsfaktors mit zwei Ausprägungen anzugeben oder ein Objekt der Klasse `factor` mit zwei Stufen, das die Ausprägungen der Variable in der ersten Stichprobe enthält. In letzterem Fall muss auch ein Faktor `y` mit zwei Stufen und derselben Länge wie `x` angegeben werden, der Daten derselben Variable aus einer zweiten Stichprobe beinhaltet.

Im Beispiel sei an je einer Stichprobe aus der Population der Frauen und der Männer erhoben worden, ob die Person raucht. Geprüft wird die H_0 , dass der Anteil der Raucher in beiden Populationen gleich ist, wobei als H_1 hier vermutet wird, dass Frauen generell häufiger rauchen. Der Aufbau der Kontingenztafel verlangt nach einem linksseitigen Test.

```
> N           <- 20                                # Stichprobengröße
> smokesFem  <- rbinom(N, size=1, p=0.6)        # Daten Frauen
> smokesMale <- rbinom(N, size=1, p=0.4)         # Daten Männer

# Faktoren, die Rauchverhalten und Geschlecht codieren
> smokes <- factor(c(smokesFem, smokesMale), labels=c("no", "yes"))
> sex     <- factor(rep(c("f", "m"), each=N))      # Faktor Geschlecht
> contT2 <- xtabs(~ sex + smokes)                 # Kontingenztafel
> addmargins(contT2)                             # Randsummen
smokes
sex   no   yes   Sum
  f    8    12    20
  m   16     4    20
Sum   24    16    40

> fisher.test(contT2, alternative="less")
Fisher's Exact Test for Count Data
data: contT2
p-value = 0.01124
alternative hypothesis: true odds ratio is less than 1
95 percent confidence interval:
0.0000000 0.6668953
sample estimates:
odds ratio
0.1751986
```

10.2.6 Kennwerte von (2×2) -Konfusionsmatrizen

(2×2) -Kontingenztafeln als Ergebnis einer zweifachen dichotomen Klassifikation (*Konfusionsmatrizen*) erlauben die Berechnung mehrerer Kennwerte, um Eigenschaften der Klassifikation

zu beschreiben. Hierzu zählen die Sensitivität, Spezifität und Relevanz sowie das odds ratio und das relative Risiko.

Als Beispiel seien die Daten herangezogen, die bereits für Fishers exakten Test auf Unabhängigkeit verwendet wurden. Dabei sollen die Abkürzungen TP (*true positive, hit*) für richtig positive, TN (*true negative*) für richtig negative, FP (*false positive*) für falsch positive und FN (*false negative, miss*) für falsch negative Diagnosen stehen.

```
> addmargins(contT1)
      diagT          #             Diagnose
disease  isHealthy  isIll  Sum    # krank? "gesund"   "krank"
       no        8     2   10    #  nein      TN      FP
       yes       1     4    5    #   ja       FN      TP
       Sum       9     6   15

> TN <- contT1[1, 1]           # true negative
> TP <- contT1[2, 2]           # true positive / hit
> FP <- contT1[1, 2]           # false positive
> FN <- contT1[2, 1]           # false negative / miss
```

Sensitivität, Spezifität und Relevanz

R stellt zur Ermittlung der folgenden Kennwerte keine eigenen Funktionen bereit, eine manuelle Berechnung ist jedoch unkompliziert. Die Prävalenz der Krankheit ist der Anteil der Kranken an der Gesamtzahl von Beobachtungen. Im Kontext des Satzes von Bayes entspricht dies der a-priori Wahrscheinlichkeit eines Merkmals.

```
> (prevalence <- sum(contT1[2, ]) / sum(contT1))
[1] 0.3333333
```

Die Sensitivität, in anderem Kontext auch *recall* genannt, ist der Quotient aus TP und der Summe von TP und FN, also das Verhältnis von richtig entdeckten zu allen zu entdeckenden Elementen. In der Sprechweise inferenzstatistischer Tests wäre dies auf theoretischer Ebene die power eines Tests. Entsprechend wäre die Wahrscheinlichkeit eines Fehlers zweiter Art (β) gleich $1 - \text{Sensitivität}$.

```
> (sensitivity <- recall <- TP / (TP+FN))
[1] 0.8
```

Die Spezifität ist der Quotient aus TN und der Summe von TN und FP, also hier das Verhältnis von richtig als gesund Eingestuften zu allen Gesunden. In der Sprechweise inferenzstatistischer Tests wäre $1 - \text{Spezifität}$ auf theoretischer Ebene gleich der Wahrscheinlichkeit eines Fehlers erster Art (α), dem somit $\frac{FP}{TN+FP}$ entspräche.

```
> (specificity <- TN / (TN+FP))
[1] 0.8
```

Die Relevanz, je nach Kontext auch als Präzision oder positiver Vorhersagewert bezeichnet, ist der Quotient aus TP und der Summe von TP und FP. Er gibt damit hier an, welcher Anteil der als krank Diagnostizierten tatsächlich krank ist. Im Kontext des Satzes von Bayes entspricht dies der a-posteriori Wahrscheinlichkeit eines Merkmals gegeben die positive Diagnose.

```
> (relevance <- precision <- TP / (TP+FP))
[1] 0.6666667
```

Der Anteil richtiger Diagnosen an allen Diagnosen wird auch als Rate der korrekten Klassifikation bezeichnet und ist der Quotient aus der Summe der Diagonalelemente und der Summe aller Elemente.

```
> (CCR <- sum(diag(contT1)) / sum(contT1))
[1] 0.8
```

Das *F*-Maß als harmonisches Mittel von Präzision und recall wird bisweilen als integriertes Gütemaß für eine Klassifikation herangezogen. Er ist nicht mit Werten einer *F*-Verteilung zu verwechseln.

```
> (Fval <- 1 / mean(1 / c(precision, recall)))
[1] 0.7272727
```

Odds ratio, Yules *Q* und relatives Risiko

Das odds ratio (OR, Chancenverhältnis) ist ein Zusammenhangsmaß für zwei dichotome Variablen und ergibt sich aus der (2×2) -Kontingenztafel ihrer gemeinsamen Häufigkeiten: $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$. Das OR berechnet sich durch Division der *Wettquotienten* (Chancen) $\frac{a}{b}$ und $\frac{c}{d}$, also als $\frac{ad}{bc}$. Repräsentieren die Zeilen der Kontingenztafel zwei Bedingungen und die Spalten die An- bzw. Abwesenheit eines Merkmals, drückt das OR die multiplikative Änderung der bedingten Verteilung des Merkmals beim Wechsel der Bedingungen aus.

Das OR lässt sich manuell, oder alternativ über `OddsRatio()` aus dem Paket `DescTools` ermitteln. Diese Funktion akzeptiert als Argument die Kontingenztafel absoluter Häufigkeiten und gibt ein über das Argument `conf.level` wählbares Konfidenzintervall für unterschiedliche Berechnungsmethoden aus. Das logarithmierte OR_{ln} ist die Differenz der logits $\ln \frac{\hat{P}_j}{1-\hat{P}_j}$, wenn \hat{P}_j die geschätzte Auftretenswahrscheinlichkeit des Merkmals in der Zeile j ist. Ist eine Zelle der empirischen Kontingenztafel gleich 0, wird vor der OR-Berechnung üblicherweise 0.5 zu allen Zellen addiert.

```
> library(DescTools)                                     # für OddsRatio()
> OddsRatio(contT1, conf.level=0.95)                  # odds ratio + Wald CI
odds ratio      lwr.ci      upr.ci
16.000000  1.092859  234.247896

> aa <- contT1[1, 1]                                    # manuelle Kontrolle
> bb <- contT1[1, 2]
> cc <- contT1[2, 1]
> dd <- contT1[2, 2]
```

```
> (OR <- (aa/bb) / (cc/dd)) # odds ratio
[1] 16
```

Auch Yules Q -Koeffizient bezieht sich auf die (2×2) -Kontingenztafel der gemeinsamen Häufigkeiten zweier dichotomer Variablen und lässt sich als Maß ihres Zusammenhangs interpretieren. Q ist mit `YuleQ()` aus dem Paket `DescTools` oder manuell als $\frac{ad-bc}{ad+bc}$ zu berechnen. Wurde das OR bereits ermittelt, gilt alternativ $Q = \frac{OR-1}{OR+1}$.

```
> library(DescTools) # für YuleQ()
> YuleQ(contT1) # Yules Q
[1] 0.8823529

> (OR-1) / (OR+1) # Kontrolle
[1] 0.882353
```

Das ebenfalls aus einer (2×2) -Kontingenztafel berechnete relative Risiko bezeichnet das Verhältnis der bedingten relativen Häufigkeiten $\frac{a}{a+b}$ und $\frac{c}{c+d}$. Im Fall der vorliegenden Daten wäre dies das Verhältnis der (auf die Zeilen) bedingten relativen Häufigkeiten, dass Gesunde bzw. Kranke als gesund diagnostiziert werden. Die Berechnung übernimmt `RelRisk()` aus dem Paket `DescTools`.

```
# "Risiko", dass Gesunde bzw. Kranke als gesund diagnostiziert werden
> (risk <- prop.table(contT1, margin=1))
  diagT
disease  isHealthy  isIll
  no        0.8     0.2
  yes       0.2     0.8

> library(DescTools) # für RelRisk()
> RelRisk(contT1) # relatives Risiko
[1] 4

> risk[1, 1] / risk[2, 1] # Kontrolle
[1] 4
```

10.2.7 ROC-Kurve und AUC

Das Paket `pROC` (Robin et al., 2011) ermöglicht die Berechnung von ROC-Kurven bei einer dichotomen Klassifikation auf Basis eines kontinuierlichen Prädiktors (vgl. Abschn. 8.1). ROC-Kurven stellen die Sensitivität gegen 1 – Spezifität der Klassifikation dar, wenn die Klassifikationsschwelle variiert. `roc(⟨AV⟩ ~ ⟨UV⟩, data=⟨Datensatz⟩)` berechnet zudem die Fläche unter der ROC-Kurve (AUC) als integriertes Gütemaß der Klassifikation. Dabei ist `⟨AV⟩` ein Faktor mit zwei Stufen, `⟨UV⟩` ein numerischer Vektor und `data` der Datensatz, aus dem beide stammen. Weitere Funktionen erlauben es, den Konfidenzbereich etwa für die Sensitivität aus Bootstrap-Replikationen (vgl. Abschn. 11.1) zu ermitteln und grafisch darzustellen (Abb. 10.2).

```

> N      <- 100                      # Stichprobengröße
> height <- rnorm(N, 175, 7)        # Körpergröße
> weight <- 0.4*height + 10 + rnorm(N, 0, 3) # Gewicht

# Median-Split Gewicht
> wFac <- cut(weight, breaks=c(-Inf, median(weight), Inf),
+               labels=c("lo", "hi"))

> regDf <- data.frame(wFac, height)      # Datensatz
> library(pROC)                         # für roc(), ci.se()
> (rocRes <- roc(wFac ~ height, data=regDf, plot=TRUE, ci=TRUE,
+                 main="ROC-Kurve", xlab="Spezifität (TN / (TN+FP))",
+                 ylab="Sensitivität (TP / (TP+FN)))")
Call:
roc.formula(formula=wFac ~ height, data=regDf, plot=TRUE, ci=TRUE)

Data: height in 50 controls (wFac lo) < 50 cases (wFac hi).
Area under the curve: 0.8148
95% CI: 0.7317-0.8979 (DeLong)

# stelle Konfidenzbereich für die Sensitivität dar
> rocCI <- ci.se(rocRes)                # CIs für Sensitivität
> plot(rocCI, type="shape")

```

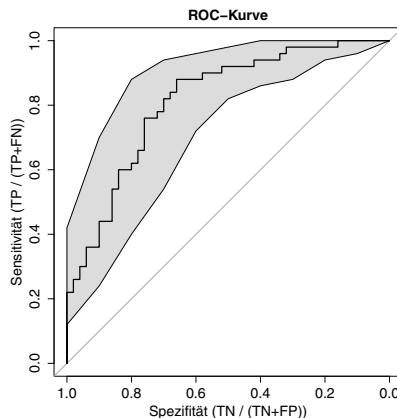


Abbildung 10.2: ROC-Kurve für eine dichotome Klassifikation samt Konfidenzbereich für die Sensitivität

10.3 Maße für Zusammenhang und Übereinstimmung

10.3.1 Zusammenhang stetiger ordinaler Variablen: Spearmans ρ und Kendalls τ

Zur Berechnung der Kovarianz und Korrelation zweier stetiger ordinaler Variablen nach Spearman bzw. nach Kendall dienen die Funktionen `cov()` und `cor()`, für die dann das Argument `method` auf "spearman" bzw. auf "kendall" zu setzen ist. Spearmans ρ ist gleich der gewöhnlichen Pearson-Korrelation beider Variablen, nachdem ihre Werte durch die zugehörigen Ränge ersetzt wurden.¹¹

```
> DV1 <- c(100, 76, 56, 99, 50, 62, 36, 69, 55, 17)
> DV2 <- c(42, 74, 22, 99, 73, 44, 10, 68, 19, -34)
> cor(DV1, DV2, method="spearman")           # Spearmans rho
[1] 0.6727273

> cor(rank(DV1), rank(DV2))                  # Korrelation der Ränge
[1] 0.6727273
```

Für Kendalls τ_a ist die Differenz der Anzahl konkordanter Paare n_{CP} und diskonkordanter Paare n_{DP} von je zwei abhängigen Messwerten zu bilden und an der Gesamtzahl möglicher Paare $\binom{n}{2} = \frac{n(n-1)}{2}$ zu relativieren. Sind X und Y die Variablen aus den abhängigen Stichproben vom Umfang n , ist ein Paar $\{(x_i, y_i), (x_j, y_j)\}$ mit $i \neq j$ dann konkordant, wenn x_i und x_j dieselbe Rangordnung aufweisen wie y_i und y_j .

Kendalls τ_b unterscheidet sich von τ_a , wenn jeweils innerhalb von X und Y identische Werte (Bindungen) vorliegen können, wobei n_{TX} die Anzahl der Bindungen in X und n_{TY} die Anzahl der Bindungen in Y angibt. Dann relativiert τ_b die Differenz $n_{CP} - n_{DP}$ am geometrischen Mittel $\sqrt{(n_{CP} + n_{DP} + n_{TX}) \cdot (n_{CP} + n_{DP} + n_{TY})}$. `cor(..., method="kendall")` berechnet bei Bindungen τ_b .

```
> cor(DV1, DV2, method="kendall")            # Kendalls tau-b
[1] 0.6

# Matrizen der paarweisen Rangvergleiche jeweils in X und Y
> cmpMat1 <- outer(DV1, DV1, ">")          # Vergleiche in X
> cmpMat2 <- outer(DV2, DV2, ">")          # Vergleiche in Y
> selMat  <- upper.tri(cmpMat1)             # relevant nur obere Dreiecksmatrix

# Anzahl konkordanter Paare -> Rangordnung in X und Y identisch
# setzt voraus, dass keine Bindungen existieren
> nCP <- sum((cmpMat1 == cmpMat2)[selMat])

# Anzahl diskonkordanter Paare -> Rangordnung in X und Y verschieden
> nDP   <- sum((cmpMat1 != cmpMat2)[selMat])    # Anzahl Versuchspersonen
> N      <- length(DV1)                         # Anzahl möglicher Paare
> nPairs <- choose(N, 2)
```

¹¹Für die polychorische und polyseriale Korrelation vgl. Abschn. 2.7.8, Fußnote 27.

```
> (tau <- (nCP-nDP) / nPairs) # Kendalls tau
[1] 0.6
```

Spearmans und Kendalls empirische Maße des Zusammenhangs lassen sich mit `cor.test()` inferenzstatistisch daraufhin prüfen, ob sie mit der H_0 verträglich sind, dass der theoretischer Zusammenhang 0 ist.

```
> cor.test(x=Vektor1, y=Vektor2, alternative=c("two.sided", "less",
+           "greater"), method=c("pearson", "kendall", "spearman"), use)
```

Die Daten beider Variablen sind als Vektoren derselben Länge über die Argumente `x` und `y` anzugeben. Ob die H_1 zwei- (`"two.sided"`), links- (negativer Zusammenhang, `"less"`) oder rechtsseitig (positiver Zusammenhang, `"greater"`) ist, legt das Argument `alternative` fest. Für den Test ordinaler Daten nach Spearman und Kendall ist das Argument `method` auf `"spearman"` bzw. `"kendall"` zu setzen. Über das Argument `use` können verschiedene Strategien zur Behandlung fehlender Werte ausgewählt werden (vgl. Abschn. 2.11.4).

```
> cor.test(DV1, DV2, method="spearman")
Spearman's rank correlation rho
data: DV1 and DV2
S = 54, p-value = 0.03938
alternative hypothesis: true rho is not equal to 0
sample estimates:
rho
0.6727273
```

Die Ausgabe des Spearman-Tests beinhaltet den Wert der Hotelling-Pabst-Teststatistik (`S`) nebst zugehörigem p -Wert (`p-value`) und Spearmans ρ (`rho`). S ergibt sich als Summe der quadrierten Differenzen zwischen den Rängen zugehöriger Messwerte in beiden Stichproben.¹²

```
> sum((rank(DV1) - rank(DV2))^2) # Teststatistik S
[1] 54
```

```
> cor.test(DV1, DV2, method="kendall") # Test nach Kendall
Kendall's rank correlation tau
data: DV1 and DV2
T = 36, p-value = 0.01667
alternative hypothesis: true tau is not equal to 0
sample estimates:
tau
0.6
```

Die Ausgabe des Kendall-Tests berichtet den Wert der Teststatistik (`T`) gemeinsam mit dem zugehörigen p -Wert (`p-value`) sowie Kendalls τ_b (`tau`). T ist hier als Anzahl konkordanter Paare definiert – die Differenz der Anzahl konkordanter und diskonkordanter Paare wäre gleichermaßen geeignet, da sich beide zur festen Anzahl möglicher Paare addieren, sofern keine Bindungen vorliegen (vgl. Fußnote 12).

¹²Die Berechnung des zugehörigen p -Wertes ist nur über eine intern definierte Funktion möglich, die Verteilungsfunktion der Teststatistik ist nicht direkt als R-Funktion vorhanden.

Goodman und Kruskals γ basiert ebenfalls auf der Anzahl konkordanter und diskonkordanter Paare. γ stimmt mit Kendalls τ_a und τ_b überein, wenn keine Rangbindungen vorliegen. Bei Bindungen unterscheidet sich γ jedoch darin, dass es die Differenz $n_{CP} - n_{DP}$ an der Summe $n_{CP} + n_{DP}$ relativiert. γ lässt sich über `GoodmanKruskalGamma()` aus dem Paket `DescTools` berechnen. Ebenfalls aus diesem Paket stammt die Funktion `SomersDelta()`, die Somers' d ausgibt – ein weiteres Zusammenhangsmaß kategorialer Variablen, das die Anzahl konkordanter und diskonkordanter Paare berücksichtigt.

10.3.2 Zusammenhang kategorialer Variablen: ϕ , Cramérs V , Kontingenzkoeffizient

Als Maße des Zusammenhangs von zwei ungeordneten kategorialen Variablen mit P bzw. mit Q Kategorien dienen

- der ϕ -Koeffizient (bei dichotomen Variablen gleich deren Korrelation),
- dessen Verallgemeinerung Cramérs V (für dichotome Variablen identisch mit $|\phi|$),
- der Pearson-Kontingenzkoeffizient CC
- und Tschuprows T (identisch zu Cramérs V für quadratische Kontingenztafeln).

Diese Kennwerte basieren auf dem χ^2 -Wert der Kontingenztafel der gemeinsamen Häufigkeiten beider Variablen. Beruht die Kontingenztafel auf n Beobachtungen, und ist $L = \min(P, Q)$ der kleinere Wert der Anzahl ihrer Zeilen und Spalten, gelten folgende Zusammenhänge: $n(L - 1)$ ist der größtmögliche χ^2 -Wert. Weiterhin gilt $\phi = \sqrt{\frac{\chi^2}{n}}$, $V = \sqrt{\frac{\chi^2}{n(L-1)}}$, $CC = \sqrt{\frac{\chi^2}{n+\chi^2}}$ und $T = \sqrt{\frac{\phi^2}{\sqrt{(P-1)\cdot(Q-1)}}}$. Die Kennwerte (und einige weitere) lassen sich mit `Assocs()` aus dem Paket `DescTools` ermitteln, Tschuprows T mit `TschuprowT()` aus demselben Paket.

```
> P      <- 2
> Q      <- 3
> DV1   <- cut(c(100, 76, 56, 99, 50, 62, 36, 69, 55, 17), breaks=P)
> DV2   <- cut(c(42, 74, 22, 99, 73, 44, 10, 68, 19, -34), breaks=Q)
> cTab  <- xtabs(~ DV1 + DV2)                      # Kontingenztafel
> N     <- sum(cTab)                                # Anzahl Beobachtungen
> library(DescTools)                               # für Assocs()
> Assocs(cTab)
            estimate lwr.ci upr.ci
Phi Coeff.        0.5477    -     -
Contingency Coeff. 0.4804    -     -
Cramer V          0.5477  0.0000 1.0000
Goodman Kruskal Gamma 0.7778  0.2679 1.0000
Kendall Tau-b     0.4950  0.0499 0.9401
Stuart Tau-c      0.5600  0.0423 1.0000
Somers D C|R       0.5600  0.0423 1.0000
Somers D R|C       0.4375  0.0296 0.8454
Pearson Correlation 0.5345 -0.1433 0.8710
Spearman Correlation 0.5217 -0.1607 0.8667
```

```

Lambda C|R           0.1667  0.0000  1.0000
Lambda R|C           0.4000  0.0000  1.0000
Lambda sym          0.2727  0.0000  0.9457
Uncertainty Coeff. C|R  0.1754 -0.0578  0.4087
Uncertainty Coeff. R|C  0.2671 -0.1107  0.6450
Uncertainty Coeff. sym 0.2118 -0.0766  0.5001
Mutual Information   0.2755      -      -

```

```

> TschuprowT(cTab)                      # Tschuprows T
[1] 0.4605779

# erwartete Zellhäufigkeiten
> expected <- outer(rowSums(cTab), colSums(cTab)) / N
> chisqVal <- sum((cTab-expected)^2 / expected)    # chi^2-Wert
> (phiVal  <- sqrt(chisqVal / N))                 # phi-Koeffizient
[1] 0.5477226

> L      <- min(dim(cTab))                  # Minimum von Zeilen, Spalten
> phisqMax <- L-1                         # maximaler phi^2-Wert
> chisqMax <- N*phisqMax                  # maximaler chi^2-Wert
> (CrV    <- sqrt(chisqVal / chisqMax))    # Cramérs V
[1] 0.5477226

> (CC <- sqrt(chisqVal / (N+chisqVal)))    # Kontingenzkoeffizient
[1] 0.4803845

> (TschT <- sqrt((chisqVal/N) / sqrt((P-1)*(Q-1))))  # Tschuprows T
[1] 0.4605779

```

Für ordinale kategoriale Variablen wird etwa von Agresti (2007, p. 229 ff.) der *linear-by-linear* Test auf Zusammenhang vorgeschlagen. Das Paket `coin` stellt für seine Umsetzung `tbl_test()` bereit.

10.3.3 Inter-Rater-Übereinstimmung

Wenn mehrere Personen (*rater*) dieselben Objekte in Kategorien einordnen oder auf einer Skala bewerten, ist häufig die Inter-Rater-Übereinstimmung bzw. -Reliabilität als Grad der Ähnlichkeit der Urteile relevant (Wirtz & Caspar, 2002). Hierbei lassen sich zum einen Fälle unterscheiden, bei denen entweder nur zwei oder auch mehr rater vorhanden sind. Zum anderen ist zu differenzieren, ob kontinuierliche Bewertungen oder ungeordnete bzw. geordnete Kategorien zum Einsatz kommen. Das im folgenden verwendete Paket `DescTools` bringt neben den beschriebenen noch weitere Funktionen zur Analyse der Inter-Rater-Übereinstimmung mit, etwa für Krippendorffs α . Die rWG Koeffizienten lassen sich mit dem `multilevel` Paket berechnen. Für den Stuart-Maxwell-Test, ob zwei rater die Kategorien einer mehrstufigen Variable mit denselben Grundwahrscheinlichkeiten verwenden, vgl. Abschn. 10.5.11.

Prozentuale Übereinstimmung

Die prozentuale Übereinstimmung der Urteile zweier oder mehr rater, die dieselben Beobachtungsobjekte in mehrere Kategorien einteilen, ist die Anzahl identischer Urteile relativiert an der Anzahl aller Urteile (multipliziert mit 100). In einer manuell mittels `xtabs()` ermittelten Kontingenztafel der gemeinsamen Häufigkeiten der Urteile stehen übereinstimmend vergebene Kategorien in der Diagonale, sofern Zeilen und Spalten dieselben Kategorien in derselben Reihenfolge umfassen. Als Beispiel diene jenes aus Bortz et al. (2010, p. 458 ff.), in dem zwei rater 100 Beobachtungsobjekte in drei Kategorien einteilen.

```
> categ <- c("V", "N", "P")                                # Rating-Kategorien
> lvls  <- factor(categ, levels=categ)                  # Kategorien als Faktor
> rtr1  <- rep(lvls, c(60, 30, 10))                      # Urteile rater 1

# Urteile rater 2
> rtr2 <- rep(rep(lvls, nlevels(lvls)), c(53,5,2, 11,14,5, 1,6,3))
> cTab <- xtabs(~ rtr1 + rtr2)                            # Kontingenztafel
> addmargins(cTab)                                         # Randsummen
      rtr2
rtr1   V   N   P   Sum
  V  53   5   2   60
  N  11  14   5   30
  P   1   6   3   10
Sum  65  25  10  100

# relative Übereinstimmung
> (agree <- sum(diag(prop.table(cTab))))
[1] 0.70
```

Im Anschluss soll das Beispiel um die Urteile eines dritten raters erweitert werden.

```
# Daten rater 3
> rtr3 <- rep(rep(lvls, nlevels(lvls)), c(48,8,3, 15,10,7, 3,4,2))

# 3D-Kontingenztafel der Urteile aller 3 Beobachter
> cTab3 <- xtabs(~ rtr1 + rtr2 + rtr3)

# Summe der Diagonalelemente der relativen Häufigkeiten
> sum(diag(apply(prop.table(cTab3), 3, diag)))
[1] 0.6
```

Cohens κ

Der Grad, mit dem die Urteile zweier rater übereinstimmen, die dieselben Beobachtungsobjekte in mehrere ungeordnete Kategorien einteilen, kann mit Cohens κ -Koeffizient ausgedrückt werden. Er zielt darauf ab, den Anteil beobachteter Übereinstimmungen mit dem Anteil auch zufällig erwartbarer Übereinstimmungen in Beziehung zu setzen. Cohens κ lässt sich

mit der Funktion `CohenKappa()` aus dem Paket `DescTools` berechnen, die als Argument die Kontingenztafel der Urteile beider rater erwartet. Mit `conf.level` lässt sich zusätzlich die Breite des Konfidenzintervalls für κ festlegen.

```
> cTab <- xtabs(~ rtr1 + rtr2)                      # Kontingenztafel
> library(DescTools)                                 # für CohenKappa()
> CohenKappa(cTab, conf.level=0.95)
  kappa      lwr.ci      upr.ci
0.4285714  0.2574917  0.5996511

# beobachteter Anteil an Übereinstimmungen (Diagonalelemente)
> fObs <- sum(diag(prop.table(cTab)))

# zufällig erwartbarer Anteil an Übereinstimmungen
> fExp    <- sum(rowSums(prop.table(cTab)) * colSums(prop.table(cTab)))
> (Ckappa <- (fObs-fExp) / (1-fExp))             # Cohens kappa
[1] 0.4285714
```

Fleiss' κ

`Kappam()` aus dem Paket `DescTools` ermittelt den κ -Koeffizienten nach Fleiss als Maß der Übereinstimmung von mehr als zwei ratern, die dieselben Objekte in mehrere ungeordnete Kategorien einteilen. Dafür ist die spaltenweise aus den Urteilen der rater zusammengestellte Matrix als Argument zu übergeben.

Als Beispiel diene jenes aus Bortz et al. (2010, p. 460 ff.), in dem sechs rater dieselben 30 Begriffe einer von fünf Kategorien (a–e) zuordnen (für das Erstellen eigener Funktionen vgl. Abschn. 15.2).

```
> rtr1 <- letters[c(4,2,2,5,2, 1,3,1,1,5, 1,1,2,1,2, 3,1,1,2,1,
+                  5,2,2,1,1, 2,1,2,1,5)]
> rtr2 <- letters[c(4,2,3,5,2, 1,3,1,1,5, 4,2,2,4,2, 3,1,1,2,3,
+                  5,4,2,1,4, 2,1,2,3,5)]
> rtr3 <- letters[c(4,2,3,5,2, 3,3,3,4,5, 4,4,2,4,4, 3,1,1,4,3,
+                  5,4,4,4,4, 2,1,4,3,5)]
> rtr4 <- letters[c(4,5,3,5,4, 3,3,3,4,5, 4,4,3,4,4, 3,4,1,4,5,
+                  5,4,5,4,4, 2,1,4,3,5)]
> rtr5 <- letters[c(4,5,3,5,4, 3,5,3,4,5, 4,4,3,4,4, 3,5,1,4,5,
+                  5,4,5,4,4, 2,5,4,3,5)]
> rtr6 <- letters[c(4,5,5,5,4, 3,5,4,4,5, 4,4,3,4,5, 5,5,2,4,5,
+                  5,4,5,4,5, 4,5,4,3,5)]

# Matrix der ratings: rater in den Spalten, Objekte in den Zeilen
> rateMat <- cbind(rtr1, rtr2, rtr3, rtr4, rtr5, rtr6)
> library(DescTools)                                # für Kappam
> Kappam(rateMat, conf.level=0.95)
  kappa      lwr.ci      upr.ci
```

```
0.4302445 0.3824725 0.4780165
```

```
# manuelle Berechnung
> nRtr <- ncol(rateMat) # Anzahl rater
> nObs <- nrow(rateMat) # Anzahl Objekte

# Vektor aller Urteile
> ratings <- c(rtr1, rtr2, rtr3, rtr4, rtr5, rtr6)

# zugehöriger Faktor, welches Objekt beurteilt wurde
> obsFac <- factor(rep(1:nObs, nRtr))

# Kontingenztafel der Häufigkeiten, wie oft jedes Objekt (Zeilen)
# einer Kategorie (Spalten) zugeordnet wurde
> cTab <- xtabs(~ obsFac + ratings)

# relative Häufigkeit jeder Beurteilungskategorie
> rateTab <- prop.table(xtabs(~ ratings))

# per Zufall erwartbarer Anteil an Übereinstimmungen
> fExp <- sum(rateTab^2)

# beobachtete paarweise Übereinstimmung pro Objekt
> fObsAll <- apply(cTab, 1, function(x) {
+           sum(x*(x-1)) / (nRtr*(nRtr-1)) } )

# mittlere beobachtete paarweise Übereinstimmung
> fObs <- mean(fObsAll)
> (Fkappa <- (fObs-fExp) / (1-fExp)) # Fleiss' kappa
[1] 0.4302445
```

Gewichtetes Cohens κ

Liegen geordnete Kategorien vor, kann nicht nur berücksichtigt werden, ob Übereinstimmung vorliegt oder nicht, sondern auch, wie stark ggf. die Diskrepanz der Urteile ist. Für die Situation mit zwei ratern stehen in `CohenKappa()` aus dem Paket `DescTools` mehrere Gewichtungen zur Verfügung: Wird das Argument `weight="Equal-Spacing"` gesetzt, gilt jede Kategorie als im selben Maße unterschiedlich zu den jeweiligen Nachbarkategorien – es soll also von gleichabständigen Kategorien ausgegangen werden.

Als Beispiel diene jenes aus Bortz et al. (2010, p. 484 ff.), in dem zwei rater die jeweils zu erwartende Einschaltquote von 100 Fernsehsendungen einschätzen.

```
# Beurteilungskategorien
> categ <- c("<10%", "11-20%", "21-30%", "31-40%", "41-50%", ">50%")
> lvls <- factor(categ, levels=categ) # Kategorien als Faktor
> tv1 <- rep(lvls, c(22, 21, 23, 16, 10, 8)) # Urteile rater 1
```

```
# Urteile rater2
> tv2 <- rep(rep(lvls, nlevels(lvls)), c(5,8,1,2,4,2, 3,5,3,5,5,0,
+           1,2,6,11,2,1, 0,1,5,4,3,3, 0,0,1,2,5,2, 0,0,1, 2,1,4))

> cTab <- xtabs(~ tv1 + tv2)                                # Kontingenztafel rater
> addmargins(cTab)                                         # Randsummen
      tv2
tv1    <10% 11-20% 21-30% 31-40% 41-50% >50% Sum
<10%     5     8     1     2     4     2   22
11-20%    3     5     3     5     5     0   21
21-30%    1     2     6    11     2     1   23
31-40%    0     1     5     4     3     3   16
41-50%    0     0     1     2     5     2   10
>50%     0     0     1     2     1     4    8
Sum       9    16    17    26    20    12  100

> library(DescTools)                                     # für CohenKappa()
> CohenKappa(cTab, weight="Equal-Spacing", conf.level=0.95)
      kappa      lwr.ci      upr.ci
0.31566846  0.07905284  0.55228407
```

Für die manuelle Berechnung muss für jede Zelle der Kontingenztafel beider rater ein Gewicht angegeben werden, das der Ähnlichkeit der von den ratern verwendeten Kategorien entspricht. Die Gewichte bilden eine Toeplitz-Matrix mit konstanten Diagonalen, wenn sie in die Kontingenztafel eingetragen werden. Übereinstimmende Kategorien (Haupt-Diagonale) haben dabei eine Ähnlichkeit von 1, die minimale Ähnlichkeit ist 0. Bei gleichabständigen Kategorien sind die Gewichte dazwischen linear zu wählen (vgl. Abschn. 15.2 für selbst erstellte Funktionen).

```
> P <- ncol(cTab)                                         # Anzahl Kategorien

# bei Unabhängigkeit erwartete Häufigkeiten
> expected <- outer(rowSums(cTab), colSums(cTab)) / sum(cTab)

# gleichabständige Gewichtungsfaktoren für Ähnlichkeit der Kategorien
> (myWeights <- seq(0, 1, length.out=P))
[1] 0.0 0.2 0.4 0.6 0.8 1.0

# Toeplitz-Matrix der Ähnlichkeits-Gewichte
> (weightsMat <- outer(1:P, 1:P, function(x, y) {
+           1 - ((abs(x-y)) / (P-1)) } ))
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
[1,] 1.0  0.8  0.6  0.4  0.2  0.0
[2,] 0.8  1.0  0.8  0.6  0.4  0.2
[3,] 0.6  0.8  1.0  0.8  0.6  0.4
[4,] 0.4  0.6  0.8  1.0  0.8  0.6
[5,] 0.2  0.4  0.6  0.8  1.0  0.8
```

```
[6,] 0.0 0.2 0.4 0.6 0.8 1.0

# Summe der gewichteten beobachteten relativen Häufigkeiten
> wfObs <- sum(cTab * weightsMat) / sum(cTab)

# Summe der gewichteten erwarteten relativen Häufigkeiten
> wfExp <- sum(expected * weightsMat) / sum(cTab)

# gewichtetes Cohens kappa
> (wKappa <- (wfObs-wfExp) / (1-wfExp))
[1] 0.3156685
```

Kendalls W

Für stetige ordinale Urteile von mehr als zwei ratern kann Kendalls W von der Funktion `KendallW()` aus dem Paket `DescTools` ermittelt werden, die als Argument eine spaltenweise aus den Urteilen der rater zusammengesetzte Matrix erwartet.¹³ Als Beispiel diene jenes aus Bortz et al. (2010, p. 466 ff.), in dem drei rater sechs Beobachtungsobjekte entsprechend der Ausprägung eines Merkmals in eine Rangreihe bringen.

```
> rtr1 <- c(1, 6, 3, 2, 5, 4) # Urteile rater 1
> rtr2 <- c(1, 5, 6, 2, 4, 3) # Urteile rater 2
> rtr3 <- c(2, 3, 6, 5, 4, 1) # Urteile rater 3
> ratings <- cbind(rtr1, rtr2, rtr3) # Matrix der Urteile
> library(DescTools) # für KendallW()
> KendallW(ratings, test=TRUE)
Kendall's coefficient of concordance W
data: ratings
Kendall chi-squared=8.5238, df=5, subjects=6, raters=3, p-value=0.1296
alternative hypothesis: W is greater than 0
sample estimates:
W
0.568254
```

Für die manuelle Berechnung wird zunächst die Matrix der Beurteiler-weisen Ränge benötigt, die im konkreten Beispiel bereits vorliegt, da Rangurteile abgegeben wurden. Bei anderen ordinalen Urteilen wäre sie aus der Matrix der ratings mit `apply(ratings, 2, rank)` zu bilden.

```
> rankMat <- ratings # Matrix der Ränge
> nObj <- nrow(rankMat) # Anzahl Objekte
> nRtr <- ncol(rankMat) # Anzahl rater
> (rankSum <- rowSums(rankMat)) # Rangsumme pro Objekt
[1] 4 14 15 9 13 8
```

¹³Der zugehörige Signifikanztest ist äquivalent zum Friedman-Test (vgl. Abschn. 10.5.7), wobei den ratern die VPn bzw. Blöcke entsprechen und den Objekten die unterschiedlichen Bedingungen.

```
# Summe der quadrierten Abweichungen der Rangsummen vom Erwartungswert
> (S <- sum((rankSum - (nRtr*(nObj+1) / 2))^2))
[1] 89.5

> (W <- (12 / (nRtr^2 * nObj*(nObj^2 - 1))) * S)      # Kendalls W
[1] 0.568254

> (chisqVal <- nRtr * (nObj-1) * W)                      # Teststatistik
[1] 8.52381

> (pVal <- pchisq(chisqVal, nObj-1, lower.tail=FALSE))    # p-Wert
[1] 0.1296329
```

Intra-Klassen-Korrelation

Die Intra-Klassen-Korrelation (ICC) wird von `ICC()` aus dem Paket `DescTools` ermittelt. Als Argument ist die Matrix der Urteile zu übergeben, wobei jede Spalte die Urteile eines raters umfasst. Die sich auf ein Objekt beziehenden Urteile befinden sich in jeweils einer Zeile.

Die ICC findet in verschiedenen Bereichen der Statistik Anwendung, u. a. eignet sie sich zur Quantifizierung der Übereinstimmung von ratern, die intervallskalierte Urteile über dieselben Objekte abgeben.¹⁴ Diese Situation lässt sich auch als Fall eines varianzanalytischen SPF- $p \cdot q$ -Designs mit einem Zwischen-Gruppen-Faktor (den ratern) und einem Intra-Gruppen-Faktor (den Objekten) betrachten (vgl. Abschn. 7.7). Unterschiedliche ICC-Varianten resultieren in Abhängigkeit davon, ob zum einen rater bzw. Objekte als feste Faktoren oder als Random-Faktoren zu betrachten sind, und ob zum anderen Einzeldaten oder bereits aggregierte Werte vorliegen.

```
> nRtr <- 4                                         # Anzahl rater
> nObs <- 6                                         # Beobachtungen pro rater
> rtr1 <- c(9, 6, 8, 7, 10, 6)                     # ratings rater 1
> rtr2 <- c(2, 1, 4, 1, 5, 2)                     # " 2
> rtr3 <- c(5, 3, 6, 2, 6, 4)                     # " 3
> rtr4 <- c(8, 2, 8, 6, 9, 7)                     # " 4
> library(DescTools)                                # für ICC()
> ICC(cbind(rtr1, rtr2, rtr3, rtr4))
Call: ICC(x = cbind(rtr1, rtr2, rtr3, rtr4))
```

Intraclass correlation coefficients

	type	ICC	F	df1	df2	p	lower bound	upper bound
Single_raters_abs	ICC1	0.17	1.79	5	18	0.16	-0.13	0.72
Single_random_raters	ICC2	0.29	11.03	5	15	0.00	0.02	0.76
Single_fixed_raters	ICC3	0.71	11.03	5	15	0.00	0.34	0.95

¹⁴Die ICC ist kein nonparametrisches Verfahren, soll aber als Maß der Inter-Rater-Reliabilität dennoch hier aufgeführt werden.

Avg_raters_absolute	ICC1k	0.44	1.79	5	18	0.16	-0.88	0.91
Avg_random_raters	ICC2k	0.62	11.03	5	15	0.00	0.07	0.93
Avg_fixed_raters	ICC3k	0.91	11.03	5	15	0.00	0.68	0.99

Die Ausgabe nennt den Wert von sechs ICC-Varianten samt der p -Werte ihres jeweiligen Signifikanztests und der Grenzen des zugehörigen zweiseitigen Vertrauensintervalls. Die Varianten ICC1–ICC3 gehen davon aus, dass jeder Wert ein Einzelurteil darstellt, während die Varianten ICC1k–ICC3k für den Fall gedacht sind, dass jeder Wert seinerseits bereits ein Mittelwert mehrerer Urteile desselben raters über dasselbe Objekt ist. Welcher ICC-Wert der zu einer konkreten Untersuchung passende ist, hat der Nutzer selbst zu entscheiden, wobei eine nähere Erläuterung über ?ICC zu finden ist.

Für die manuelle Berechnung sind Effekt- und Residual-Quadratsummen aus zwei Varianzanalysen notwendig: zum einen aus der einfaktoriellen ANOVA (vgl. Abschn. 7.3) mit nur den Objekten als UV, zum anderen aus der zweifaktoriellen ANOVA (vgl. Abschn. 7.5) mit Objekten und ratern als Faktoren.

```
> ratings <- c(rtr1, rtr2, rtr3, rtr4) # alle ratings

# zugehöriger Faktor, von welchem rater ein rating stammt
> rateFac <- factor(rep(paste("rater", 1:nRtr, sep=""), each=nObs))

# zugehöriger Faktor, zu welchem Objekt ein rating gehört
> obsFac <- factor(rep(paste("obj", 1:nObs, sep=""), times=nRtr))

# Varianzanalyse nur mit Faktor Beobachtungsobjekt
> (anObs <- anova(lm(ratings ~ obsFac))) # ...

# zweifaktorielle Varianzanalyse: Beobachtungsobjekt und rater
> (anBoth <- anova(lm(ratings ~ obsFac + rateFac))) # ...

# relevante mittlere Quadratsummen extrahieren
> MSobs <- anObs["obsFac", "Mean Sq"] # Effekt Objekt 1-fakt. ANOVA
> MSEobs <- anObs["Residuals", "Mean Sq"] # Residuen 1-fakt. ANOVA
> MSrtr <- anBoth["rateFac", "Mean Sq"] # Effekt rater 2-fakt. ANOVA
> MSEboth <- anBoth["Residuals", "Mean Sq"] # Residuen 2-fakt. ANOVA

> (ICC1 <- (MSobs-MSEobs) / (MSobs + (nRtr-1)*MSEobs))
[1] 0.1657418

> (ICC2 <- (MSobs-MSEboth) / (MSobs + (nRtr-1)*MSEboth
+ ((nRtr/nObs) * (MSrtr - MSEboth))))
[1] 0.2897638

> (ICC3 <- (MSobs-MSEboth) / (MSobs + (nRtr-1)*MSEboth))
[1] 0.7148407

> (ICC1k <- (MSobs-MSEobs) / (MSobs))
```

```
[1] 0.4427971

> (ICC2k <- (MSobs-MSEboth) / (MSobs + ((1/nObs) * (MSrtr-MSEboth))))
[1] 0.6200505

> (ICC3k <- (MSobs-MSEboth) / MSobs)
[1] 0.9093155
```

10.4 Tests auf gleiche Variabilität

Die in Abschn. 10.5.4 und 10.5.6 vorgestellten Tests prüfen, ob eine Variable in zwei oder mehr unabhängigen Bedingungen denselben Median besitzt. Dafür setzen sie u. a. voraus, dass die Variable in allen Bedingungen dieselbe Variabilität hat, die Verteilungen also gleich breit sind. Wird untersucht, ob die erhobenen Werte mit dieser Annahme konsistent sind, ist zu berücksichtigen, dass i. d. R. die H_0 den gewünschten Zustand darstellt. Um die power der Tests zu vergrößern, wird mitunter ein höher als übliches α -Niveau in der Größenordnung von 0.2 gewählt.¹⁵

10.4.1 Mood-Test

Mit dem Mood-Test lässt sich prüfen, ob die empirische Variabilität einer stetigen ordinalen Variable in zwei unabhängigen Stichproben mit der H_0 verträglich ist, dass die Variable in beiden Bedingungen dieselbe theoretische Variabilität besitzt.¹⁶

```
> mood.test(x=Vektor, y=Vektor,
+             alternative=c("two.sided", "less", "greater"))
```

Unter x und y sind die Daten aus beiden Stichproben einzutragen. Alternativ zu x und y kann auch eine Modellformel $\langle AV \rangle \sim \langle UV \rangle$ eingegeben werden. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter data zu nennen. Das Argument alternative bezieht sich auf die Variabilität von x im Vergleich zu jener von y.

```
> DV1 <- c(12, 13, 29, 30)                                # Daten Gruppe 1
> DV2 <- c(15, 17, 18, 24, 25, 26)                          # Daten Gruppe 2
> Nj  <- c(length(DV1), length(DV2))                      # Gruppengrößen
> DV  <- c(DV1, DV2)                                       # Gesamt-Daten
> IV  <- factor(rep(1:2, Nj), labels=c("A", "B"))        # Gruppenzugehörigkeit
> mood.test(DV ~ IV, alternative="greater")
Mood two-sample test of scale
data: DV by IV
Z = 2.6968, p-value = 0.003500
alternative hypothesis: greater
```

¹⁵Für die zugehörigen parametrischen Tests und den Fall mit mehr als zwei Gruppen vgl. Abschn. 7.1.

¹⁶Vorauszusetzen ist, dass die Verteilung in beiden Stichproben denselben theoretischen Median hat. Bei Zweifeln daran können die Daten zuvor gruppenweise zentriert werden, indem man von jedem Wert den Gruppenmedian abzieht.

Die Ausgabe des Tests umfasst den Wert der z -transformierten Teststatistik (Z), wofür Erwartungswert und Varianz einer asymptotisch gültigen Normalverteilung verwendet werden. Der p -Wert wird unter `p-value` ausgegeben. Das Ergebnis lässt sich manuell prüfen, wobei auf die enge Verwandtschaft zum Wilcoxon-Rangsummentest hingewiesen sei (vgl. Abschn. 10.5.4): Lediglich die Wahl der Gewichte ist hier eine andere.

```
# Gewichte: quadrierte Differenzen der Ränge zu ihrem Mittelwert
> gX <- (rank(DV) - mean(rank(DV)))^2
> MN <- sum(gX[IV == "A"])           # Teststat.: Summe Gewichte der 1. Gruppe

# Erwartungswert und theoretische Varianz der Teststatistik
> N      <- sum(Nj)                  # Gesamtzahl VPn
> muMN   <- (Nj[1] * (N^2 - 1)) / 12    # Erwartungswert
> varMN <- (Nj[1]*Nj[2] * (N+1) * (N^2 - 4)) / 180    # Varianz
> (MNz  <- (MN-muMN) / sqrt(varMN))        # theo. z-Transformation
[1] 2.696799

> (pVal <- pnorm(MNz, 0, 1, lower.tail=FALSE))  # p-Wert einseitig
[1] 0.003500471
```

10.4.2 Ansari-Bradley-Test

Der Ansari-Bradley-Test stellt eine Alternative zum Mood-Test dar und ist ebenfalls für den Vergleich der Variabilität von stetigen, ordinalen Daten aus zwei unabhängigen Stichproben geeignet (vgl. Fußnote 16).

```
> ansari.test(x=<Vektor>, y=<Vektor>, exact=NULL,
+               alternative=c("two.sided", "less", "greater"))
```

Unter `x` und `y` sind die Daten aus beiden Stichproben einzutragen. Alternativ zu `x` und `y` kann auch eine Modellformel `AV ~ UV` eingegeben werden. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Das Argument `alternative` bezieht sich auf die Variabilität `x` im Vergleich zu jener von `y`. Für die (bei großen Stichproben u. U. etwas zeitaufwendigere) Berechnung auf Basis der exakten anstatt asymptotisch gültigen Normalverteilung ist `exact=TRUE` zu setzen. Das folgende Beispiel verwendet die Daten aus Abschn. 10.4.1.

```
> ansari.test(DV ~ IV, alternative="greater", exact=FALSE)
Ansari-Bradley test
data: DV by IV
AB = 6, p-value = 0.004687
alternative hypothesis: true ratio of scales is greater than 1
```

Die Ausgabe des Tests umfasst den Wert der Teststatistik (`AB`) sowie den zugehörigen p -Wert (`p-value`). Das Ergebnis lässt sich manuell bestätigen, wobei sich die Teststatistik lediglich in der Wahl der Gewichte von jener des Mood-Tests unterscheidet.

```
# Gewichtungsfaktoren basieren auf abs. Abweichung der Ränge zu ihrem Mittel
> gX <- mean(rank(DV)) - abs(rank(DV) - mean(rank(DV)))
> (AB <- sum(gX[IV == "A"]))      # Teststat.: Summe Gewichte der 1. Gruppe
[1] 6

# Erwartungswert und theoretische Varianz der Teststatistik
# abhängig davon, ob Gesamt-N gerade oder ungerade ist
> muABe <- (Nj[1] * (N+2)) / 4                                # gerades N
> muABo <- (Nj[1] * (N+1)^2) / (4*N)                            # ungerades N
> varABe <- (Nj[1]*Nj[2] * (N^2 - 4)) / (48 * (N-1))          # gerades N
> varABo <- (Nj[1]*Nj[2] * (N+1) * (3 + N^2)) / (48*N^2)       # ungerades N
> muAB <- ifelse(N %% 2, muABo, muABe)                         # hier passender EW
> varAB <- ifelse(N %% 2, varABo, varABe)                       # hier passende Varianz
> ABz <- (AB-muAB) / sqrt(varAB)                               # theo. z-Transformation
> pnorm(ABz, 0, 1)                                              # p-Wert linksseitig
[1] 0.004687384
```

10.5 Tests auf Übereinstimmung von Verteilungen

Die im folgenden aufgeführten Tests lassen sich mit Werten von stetigen ordinalen (aber nicht notwendigerweise metrischen und normalverteilten) Variablen durchführen und testen Hypothesen über die Lage ihrer Verteilungen.

10.5.1 Kolmogorov-Smirnov-Test für zwei Stichproben

Der Kolmogorov-Smirnov-Test prüft die Daten einer stetigen Variable aus zwei unabhängigen Stichproben daraufhin, ob sie mit der Nullhypothese verträglich sind, dass die Variable in beiden Bedingungen dieselbe Verteilung besitzt (für den Anpassungstest vgl. Abschn. 10.1.3). Als *Omnibus*-Test prüft er gleichzeitig, ob Lage und Form beider Verteilungen übereinstimmen. Dazu wird sowohl für das Argument `x` von `ks.test()` als auch für `y` ein Datenvektor angegeben. Über das Argument `alternative` sind ungerichtete wie gerichtete Alternativhypotesen prüfbar, wobei sich letztere darauf beziehen, ob `y` stochastisch kleiner ("`less`") oder größer ("`greater`") als `x` ist (vgl. Abschn. 10.1.3, Fußnote 4).

```
> DV1 <- round(rnorm(8, mean=1, sd=2), 2)                  # Daten Stichprobe 1
> DV2 <- round(rnorm(8, mean=3, sd=2), 2)                  # Daten Stichprobe 2
> ks.test(DV1, DV2, alternative="greater")
Two-sample Kolmogorov-Smirnov test
data: DV1 and DV2
D^+ = 0.5, p-value = 0.1353
alternative hypothesis: the CDF of x lies above that of y
```

Die Teststatistiken werden wie im Anpassungstest gebildet, wobei alle möglichen absoluten Differenzen zwischen den kumulierten relativen Häufigkeiten der Daten beider Stichproben in die Teststatistiken einfließen (Abb. 10.3).

```

> sortBoth <- sort(c(DV1, DV2))      # kombinierte sortierte Daten
> both1    <- ecdf(DV1)(sortBoth)    # kumulierte rel. Häufigkeiten 1
> both2    <- ecdf(DV2)(sortBoth)    # kumulierte rel. Häufigkeiten 2
> diff1    <- both1 - both2        # Differenzen 1
> diff2    <- c(0, both1[-length(both1)]) - both2      # Differenzen 2
> diffBoth <- c(diff1, diff2)       # kombinierte Differenzen
> (DtwoS   <- max(abs(diffBoth)))  # Teststatistik zweiseitig
[1] 0.5

> (Dless   <- abs(min(diffBoth)))  # Teststatistik "less"
[1] 0

> (Dgreat  <- abs(max(diffBoth)))  # Teststatistik "greater"
[1] 0.5

# Kontrolle über Ausgabe von ks.test()
> ks.test(DV1, DV2, alternative="less")$statistic
D^-
 0

> ks.test(DV1, DV2, alternative="two.sided")$statistic
D
0.5

# grafische Darstellung: lege Wertebereich der x-Achse fest
> xRange <- c(min(sortBoth)-1, max(sortBoth)+1)

# zeichne kumulierte relative Häufigkeiten 1 ein
> plot(ecdf(DV1), xlim=xRange, main="Kolmogorov-Smirnov-Test für zwei
+     Stichproben", xlab=NA, col.points="blue", col.hor="blue", lwd=2)

# füge kumulierte relative Häufigkeiten 2 hinzu
> par(new=TRUE)
> plot(ecdf(DV2), xlim=xRange, main=NA, xaxt="n", xlab=NA, ylab=NA,
+     lwd=2, col.points="red", col.hor="red")

# zeichne alle möglichen Abweichungen ein
> X   <- rbind(sortBoth, sortBoth)
> Y1  <- rbind(both1, both2)
> Y2  <- rbind(c(0, both1[1:(length(both1)-1)]), both2)
> matlines(X, Y1, col="darkgray", lty=1, lwd=2)
> matlines(X, Y2, col="darkgray", lty=1, lwd=2)
> legend(x="bottomright", legend=c("kumulierte relative Häufigkeiten 1",
+     "kumulierte relative Häufigkeiten 2"), col=c("blue", "red"), lwd=2)

```

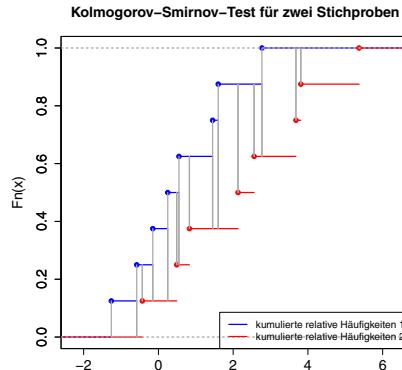


Abbildung 10.3: Kolmogorov-Smirnov-Test für zwei Stichproben: Abweichungen zwischen kumulierten relativen Häufigkeiten der Daten aus beiden Bedingungen.

10.5.2 Vorzeichen-Test

Der Vorzeichen-Test für den Median ist anwendbar, wenn eine Variable eine symmetrische Verteilung unbekannter Form besitzt. Es wird die H_0 getestet, dass der Median der Verteilung gleich einem bestimmten Wert m_0 ist.¹⁷ Sowohl gerichtete wie ungerichtete Alternativhypotesen sind möglich. Eine Umsetzung liefert `SignTest()` aus dem Paket `DescTools`.

```
> SignTest(<Vektor>, mu=(Median unter H0), conf.level=0.95,
+           alternative=c("two.sided", "less", "greater"))
```

Neben dem Vektor der Daten ist für das Argument `mu` der Median unter H_0 anzugeben. Die Argumente `alternative` und `conf.level` beziehen sich auf die Größe des Medians unter H_1 im Vergleich zu seiner Größe unter H_0 bzw. auf die Breite seines Vertrauensintervalls.

```
> medH0 <- 30                                # Median unter H0
> DV     <- sample(0:100, 20, replace=TRUE)    # Daten
> library(DescTools)                          # für SignTest()
> SignTest(DV, mu=medH0)
One-sample Sign-Test
data:  DV
S = 15, number of differences = 20, p-value = 0.04139
alternative hypothesis: true median is not equal to 30
95.9 percent confidence interval:
 34 74
sample estimates:
median of the differences
45
```

¹⁷Bei Variablen mit symmetrischer Verteilung und eindeutig bestimmtem Median ist dieser gleich dem Erwartungswert, sofern letzterer existiert.

Für die manuelle Kontrolle werden zunächst aus der Stichprobe diejenigen Werte eliminiert, die gleich m_0 sind.¹⁸ Teststatistik ist die Anzahl der beobachteten Werte $> m_0$ – bei intervallskalierten Daten wird also die Anzahl positiver Differenzen der Werte zu m_0 gezählt. Diese Anzahl besitzt unter H_0 eine Binomialverteilung mit der Trefferwahrscheinlichkeit 0.5. Um den p -Wert der Teststatistik zu berechnen, dient `pbinom()` für die Verteilungsfunktion der Binomialverteilung. Da die Binomialverteilung unter H_0 symmetrisch ist, kann der p -Wert des zweiseitigen Binomialtests als das Doppelte des einseitigen p -Wertes gebildet werden.

```

> N      <- length(DV)                                # Anzahl an Beobachtungen
> DV    <- DV[DV != medH0]                          # Werte = medH0 eliminieren
> (obs <- sum(DV > medH0))                      # Teststatistik
[1] 15

# rechtsseitiger Test
> (pGreater <- pbinom(obs-1, N, 0.5, lower.tail=FALSE))
[1] 0.02069473

> (pTwoSided <- 2 * pGreater)
[1] 0.04138947

```

Der Vorzeichentest lässt sich ebenso auf Daten einer Variable aus zwei abhängigen Stichproben anwenden, deren zugehörige Verteilungen darauf getestet werden sollen, ob ihre Lageparameter identisch sind. Hierfür ist ebenfalls `SignTest()` geeignet, wobei die Funktion intern die paarweisen Differenzen bildet und wie beschrieben mit $m_0 = 0$ testet.

10.5.3 Wilcoxon-Vorzeichen-Rang-Test für eine Stichprobe

Der Wilcoxon-Vorzeichen-Rang-Test prüft, ob die in einer Stichprobe ermittelten Werte einer Variable mit symmetrischer Verteilung mit der H_0 verträglich sind, dass der Median der Verteilung gleich einem bestimmten Wert m_0 ist. Anders als beim t -Test für eine Stichprobe (vgl. Abschn. 7.2.1) wird nicht vorausgesetzt, dass die Variable normalverteilt ist. Im Vergleich zum Vorzeichen-Test für den Median wird neben der Anzahl der Werte $> m_0$ auch das Ausmaß ihrer Differenz zu m_0 berücksichtigt.

```

> wilcox.test(x=<Vektor>, alternative=c("two.sided", "less", "greater"),
+               mu=0, conf.int=FALSE)

```

Unter `x` ist der Datenvektor einzutragen. Mit `alternative` wird festgelegt, ob die H_1 bzgl. des Vergleichs mit dem unter H_0 angenommenen Median `mu` gerichtet oder ungerichtet ist. `"less"` und `"greater"` beziehen sich dabei auf die Reihenfolge Median unter H_1 `"less"` bzw. `"greater"` m_0 . Um ein Konfidenzintervall für den Median zu erhalten, ist `conf.int=TRUE` zu setzen.

¹⁸ Auch andere Vorgehensweisen werden diskutiert, insbesondere wenn es viele Werte gleich m_0 gibt. So können im Fall geradzahlig vieler Werte gleich m_0 die Hälfte dieser Werte als $< m_0$, die andere Hälfte als $> m_0$ codiert werden. Im Fall ungeradzahlig vieler Werte gleich m_0 wird ein Wert eliminiert und dann wie für geradzahlig viele Werte beschrieben verfahren.

Als Beispiel diene jenes aus Büning und Trenkler (1994, p. 90 ff.): An Studenten einer Fachrichtung sei der IQ-Wert erhoben worden. Diese Werte sollen daraufhin geprüft werden, ob sie mit der H_0 verträglich sind, dass der theoretische Median 110 beträgt. Unter H_1 sei der Median größer.

```
> IQ      <- c(99, 131, 118, 112, 128, 136, 120, 107, 134, 122)
> medH0 <- 110                                # Median unter H0
> wilcox.test(IQ, alternative="greater", mu=medH0, conf.int=TRUE)
Wilcoxon signed rank test
data: IQ
V = 48, p-value = 0.01855
alternative hypothesis: true location is greater than 110
95 percent confidence interval:
113.5 Inf
sample estimates:
(pseudo)median
121
```

Die Ausgabe liefert die Teststatistik (V) und den p -Wert (`p-value`). Das Ergebnis lässt sich auch manuell nachvollziehen. Die diskret verteilte Teststatistik berechnet sich dabei als Summe der Ränge der absoluten Differenzen zu m_0 , wobei nur die Ränge von Werten aufsummiert werden, die $> m_0$ sind. Um den p -Wert der Teststatistik zu erhalten, dient `psignrank()` als Verteilungsfunktion der Teststatistik (vgl. Abschn. 5.3.2, Fußnote 7). Wurde `conf.int=TRUE` gesetzt, enthält die Ausgabe neben dem Konfidenzintervall für den Median unter `(pseudo)median` den zugehörigen Hodges-Lehmann-Schätzer (vgl. Abschn. 2.7.4).

```
> (diffIQ <- IQ-medH0)                      # Differenzen zum Median unter H0
[1] -11 21 8 2 18 26 10 -3 24 12

> (idx <- diffIQ > 0)                      # Wert > Median unter H0?
[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE

# Ränge der absoluten Differenzen zum Median unter H0
> (rankDiff <- rank(abs(diffIQ)))
[1] 5 8 3 1 7 10 4 2 9 6

# Teststatistik: Summe über die Ränge von Werten > Median unter H0
> (V <- sum(rankDiff[idx]))
[1] 48

# p-Wert einseitig
> (pVal <- psignrank(V-1, length(IQ), lower.tail=FALSE))
[1] 0.01855469

# Hodges-Lehmann-Schätzer: Median der Mittelwerte aller Wertepaare
> pairM <- outer(IQ, IQ, FUN="+") / 2        # Mittelwerte aller Paare
> (HLloc <- median(pairM[lower.tri(pairM, diag=TRUE)]))    # deren Median
[1] 121
```

10.5.4 Wilcoxon-Rangsummen-Test / Mann-Whitney-U-Test für zwei unabhängige Stichproben

Im Wilcoxon-Rangsummen-Test für unabhängige Stichproben werden die in zwei Stichproben ermittelten Werte einer Variable daraufhin miteinander verglichen, ob sie mit der H_0 verträglich sind, dass die Verteilungen der Variable in den zugehörigen Bedingungen identisch sind. Anders als im analogen *t*-Test (vgl. Abschn. 7.2.2) wird nicht vorausgesetzt, dass die Variable in den Gruppen normalverteilt ist. Gefordert wird jedoch, dass die Verteilungen in ihrer Form in beiden Bedingungen übereinstimmen, d. h. lediglich ggf. horizontal verschobene Versionen voneinander sind. Es kann sowohl gegen eine ungerichtete wie gerichtete H_1 getestet werden, die sich auf den Lageparameter der Verteilungen (etwa den Median) bezieht.

```
> wilcox.test(x=<Vektor>, y=<Vektor>, paired=FALSE, conf.int=FALSE,
+               alternative=c("two.sided", "less", "greater"))
```

Unter *x* sind die Daten der ersten Stichprobe einzutragen, unter *y* entsprechend die der zweiten. Alternativ zu *x* und *y* kann auch eine Modellformel $\langle AV \rangle \sim \langle UV \rangle$ angegeben werden. Dabei ist $\langle UV \rangle$ ein Gruppierungsfaktor derselben Länge wie $\langle AV \rangle$ und gibt für jede Beobachtung in $\langle AV \rangle$ die zugehörige UV-Stufe an. Geschieht dies mit Variablen, die aus einem Datensatz stammen, muss dieser unter *data* eingetragen werden. Mit *alternative* wird festgelegt, ob die H_1 gerichtet oder ungerichtet ist. "less" und "greater" beziehen sich dabei auf den Lageparameter in der Reihenfolge *x* "less" bzw. "greater" *y*. Mit dem Argument *paired* wird angegeben, ob es sich um unabhängige (FALSE) oder abhängige (TRUE) Stichproben handelt. Um ein Konfidenzintervall für die Differenz der Lageparameter zu erhalten, ist *conf.int*=TRUE zu setzen.

Als Beispiel diene jenes aus Bortz et al. (2010, p. 201 ff.): Es wird vermutet, dass sich durch die Einnahme eines Medikaments die Reaktionszeit im Vergleich zu einer Kontrollgruppe verkürzen lässt.

```
> rtCtrl <- c(85,106,118,81,138,90,112,119,107,95,88,103)
> rtDrug <- c(96,105,104,108,86,84,99,101,78,124,121,97,129,87,109)
> Nj      <- c(length(rtCtrl), length(rtDrug))    # Gruppengrößen

# Faktor der Gruppenzugehörigkeiten
> IV      <- factor(rep(1:2, Nj), labels=c("control", "drug"))
> rtAll <- c(rtCtrl, rtDrug)                      # Gesamtstichprobe
> wilcox.test(rtAll ~ IV, alternative="greater", conf.int=TRUE)

Wilcoxon rank sum test
data: rtAll by IV
W = 94, p-value = 0.4333
alternative hypothesis: true location shift is greater than 0
95 percent confidence interval:
-11 Inf
sample estimates:
difference in location
2
```

Die Ausgabe nennt den Wert der Teststatistik (*W*) gefolgt vom *p*-Wert (*p-value*). Das Ergebnis lässt sich auch manuell nachvollziehen. Dazu muss für die Daten der ersten Stichprobe die

Summe ihrer Ränge in der Gesamtstichprobe ermittelt werden. Die diskret verteilte Teststatistik ergibt sich, indem von dieser Rangsumme ihr theoretisches Minimum $\sum_{i=1}^{n_1} i = \frac{n_1 n_1 + 1}{2}$ abgezogen wird. Um den *p*-Wert zu erhalten, dient `pwilcox()` als Verteilungsfunktion der Teststatistik (vgl. Abschn. 5.3.2, Fußnote 7). Wurde `conf.int=TRUE` gesetzt, enthält die Ausgabe neben dem Konfidenzintervall für die Differenz der Lageparameter unter `difference in location` den zugehörigen Hodges-Lehmann-Schätzer (vgl. Abschn. 2.7.4).

```
> gX <- rank(rtAll) # Gewichte = Ränge
> (W <- sum(gX[IV == "control"]) - sum(1:Nj[1])) # Teststatistik
[1] 94

# p-Wert einseitig
> (pVal <- pwilcox(W-1, Nj[1], Nj[2], lower.tail=FALSE))
[1] 0.433342

# Hodges-Lehmann-Schätzer der Differenz der Lageparameter
> pairD <- outer(rtCtrl, rtDrug, "-") # alle paarweisen Differenzen
> (HLDl <- median(pairD)) # deren Median
[1] 2
```

Die Funktion `pairwise.wilcox.test()` dient dazu, simultan alle paarweisen Vergleiche zwischen jeweils zwei von insgesamt mehreren Gruppen mit dem Wilcoxon-Rangsummen-Test durchzuführen. Sie bietet mit dem Argument `p.adjust.method` verschiedene Möglichkeiten zur Adjustierung des α -Niveaus.

Der Mann-Whitney-*U*-Test zählt für jeden Wert der ersten Stichprobe, wie viele Werte der zweiten Stichprobe kleiner als er sind. Die Teststatistik *U* ist die Summe dieser Häufigkeiten und führt zum selben Wert wie die oben definierte Teststatistik des Wilcoxon-Tests, besitzt also auch dieselbe Verteilung.

```
> (U <- sum(outer(rtCtrl, rtDrug, ">=")))
[1] 94
```

10.5.5 Wilcoxon-Test für zwei abhängige Stichproben

Der Wilcoxon-Test für zwei abhängige Stichproben wird wie jener für unabhängige Stichproben durchgeführt, jedoch ist hier das Argument `paired=TRUE` von `wilcox.test()` zu verwenden. Der Test setzt voraus, dass sich die in `x` und `y` angegebenen Daten einander paarweise zuordnen lassen, weshalb `x` und `y` dieselbe Länge besitzen müssen. Nach Bildung der paarweisen Differenzen einander zugeordneter Werte wird im Wilcoxon-Vorzeichen-Rang-Test für eine Stichprobe die H_0 getestet, dass der theoretische Median der Differenzwerte gleich 0 ist.

10.5.6 Kruskal-Wallis-*H*-Test für unabhängige Stichproben

Der Kruskal-Wallis-*H*-Test verallgemeinert die Fragestellung eines Wilcoxon-Tests auf Situationen, in denen Werte einer Variable in mehr als zwei unabhängigen Stichproben ermittelt wurden. Unter H_0 sind die Verteilungen der Variable in den zugehörigen Bedingungen identisch. Die

unspezifische H_1 besagt, dass sich mindestens zwei Lageparameter unterscheiden. Anders als in der einfaktoriellen Varianzanalyse (vgl. Abschn. 7.3) wird nicht vorausgesetzt, dass die Variable in den Bedingungen normalverteilt ist. Gefordert wird jedoch, dass die Form der Verteilungen in allen Bedingungen übereinstimmt, die Verteilungen also lediglich ggf. horizontal verschobene Versionen voneinander darstellen.

```
> kruskal.test(formula=<Modellformel>, data=<Datensatz>, subset=<Indexvektor>)
```

Unter **formula** sind Daten und Gruppierungsvariable als Modellformel $\langle AV \rangle \sim \langle UV \rangle$ zu nennen, wobei $\langle UV \rangle$ ein Gruppierungsfaktor derselben Länge wie $\langle AV \rangle$ ist und für jede Beobachtung in $\langle AV \rangle$ die zugehörige UV-Stufe angibt. Geschieht dies mit Variablen, die aus einem Datensatz stammen, muss dieser unter **data** eingetragen werden. Das Argument **subset** erlaubt es, nur eine Teilmenge der Fälle einfließen zu lassen – es erwartet einen entsprechenden Indexvektor, der sich auf die Zeilen des Datensatzes bezieht.

Als Beispiel diene jenes aus Büning und Trenkler (1994, p. 183 ff.): Es wird vermutet, dass der IQ-Wert in vier Studiengängen einen unterschiedlichen Erwartungswert besitzt.

```
# IQ-Werte in den einzelnen Studiengängen
> IQ1 <- c(99, 131, 118, 112, 128, 136, 120, 107, 134, 122)
> IQ2 <- c(134, 103, 127, 121, 139, 114, 121, 132)
> IQ3 <- c(120, 133, 110, 141, 118, 124, 111, 138, 120)
> IQ4 <- c(117, 125, 140, 109, 128, 137, 110, 138, 127, 141, 119, 148)
> DV  <- c(IQ1, IQ2, IQ3, IQ4)                                # kombinierte Daten

# Stichprobengrößen
> Nj <- c(length(IQ1), length(IQ2), length(IQ3), length(IQ4))
> N   <- sum(Nj)                                              # Gesamt-N

# Faktor der Gruppenzugehörigkeiten
> IV  <- factor(rep(1:4, Nj), labels=c("I", "II", "III", "IV"))
> KWdf <- data.frame(IV, DV)                                    # Datensatz
> kruskal.test(DV ~ IV, data=KWdf)

Kruskal-Wallis rank sum test
data: DV by IV
Kruskal-Wallis chi-squared = 1.7574, df = 3, p-value = 0.6242
```

Die Ausgabe nennt den Wert der asymptotisch χ^2 -verteilten H -Teststatistik gefolgt von den Freiheitsgraden (**df**) und dem p -Wert (**p-value**). Das Ergebnis lässt sich auch manuell nachvollziehen, wobei das zentrale Element der Teststatistik das Quadrat der pro Gruppe gebildeten Summe der Ränge in der Gesamtstichprobe ist, das jeweils an der zugehörigen Gruppengröße relativiert wird.¹⁹

```
> (rankSumI <- tapply(rank(DV), IV, sum))                  # Rangsumme pro Gruppe
      I      II      III      IV
    168.5   160.0   173.0   278.5
```

¹⁹Im gewählten Beispiel sind die Ränge nicht eindeutig, es treten also Bindungen auf. Für diesen Fall gibt **rank()** in der Voreinstellung mittlere Ränge aus, was vom Vorgehen in **kruskal.test()** abweicht. Die manuell berechnete Teststatistik und p -Wert stimmen deshalb nicht exakt mit jenen aus **kruskal.test()** überein.

```
> (H <- (12 / (N*(N+1))) * sum(rankSumI^2/Nj) - 3*(N+1)) # Teststatistik
[1] 1.75531

> (pVal <- pchisq(H, nlevels(IV)-1, lower.tail=FALSE))      # p-Wert
[1] 0.624708
```

Für den Jonckheere-Terpstra-Trend-Test für geordnete Gruppen vgl. `JonckheereTerpstraTest()` aus dem Paket `DescTools`.

10.5.7 Friedman-Rangsummen-Test für abhängige Stichproben

Der Rangsummentest nach Friedman dient der Analyse von Daten einer Variable, die in p abhängigen Stichproben erhoben wurde. Jede Menge von p abhängigen Beobachtungen (eine aus jeder Bedingung) wird dabei als Block bezeichnet und stammt entweder vom selben Beobachtungsobjekt (Messwiederholung) oder von mehreren homogenen, also gematchten Beobachtungsobjekten. Wie im Kruskal-Wallis- H -Test wird die H_0 geprüft, dass die Verteilung der Variable in allen Bedingungen identisch ist. Die unspezifische H_1 besagt, dass sich mindestens zwei Lageparameter unterscheiden. Anders als in der einfaktoriellen Varianzanalyse für abhängige Gruppen (vgl. Abschn. 7.4) wird nicht vorausgesetzt, dass die blockweise als Vektor zusammengefasste Variable gemeinsam normalverteilt ist.

```
> friedman.test(y=<Daten>, groups=<Faktor>, blocks=<Faktor>)
```

Die Daten müssen im Long-Format vorliegen (vgl. Abschn. 3.3.9): Unter `y` ist der Vektor aller Daten anzugeben. Als zweites Argument wird für `groups` ein Faktor derselben Länge wie `y` übergeben, der die Gruppenzugehörigkeit jeder Beobachtung in `y` codiert. `blocks` ist ebenfalls ein Faktor derselben Länge wie `y` und gibt für jede Beobachtung in `y` an, zu welchem Block sie gehört – z. B. von welcher VP sie stammt, wenn Messwiederholung vorliegt. Alternativ lässt sich auch eine Modellformel der Form `y ~ groups | blocks` mit denselben Bedeutungen nennen. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, muss dieser für das Argument `data` eingetragen werden.

Als Beispiel diene jenes aus Bortz et al. (2010, p. 269 ff.): An Ratten wird die Auswirkung von zentralnervös anregenden Präparaten erhoben, wobei die Anzahl der pro Zeiteinheit gemessenen Umdrehungen in einem Laufrad die AV ist. Aus jeweils vier hinsichtlich verschiedener Störvariablen homogenen Ratten werden fünf Blöcke gebildet und jeder Block in allen vier Bedingungen beobachtet.

```
# Daten in den einzelnen Bedingungen
> DVcaff <- c(14, 13, 12, 11, 10)          # Koffein
> DVds   <- c(11, 12, 13, 14, 15)          # Medikament einfache Dosis
> DVdd   <- c(16, 15, 14, 13, 12)          # Medikament doppelte Dosis
> DVplac <- c(13, 12, 11, 10, 9)           # Placebo
> DV     <- c(DVcaff, DVds, DVdd, DVplac) # Gesamt-Daten
> nBl    <- length(DVcaff)                  # Anzahl Blöcke
> P      <- 4                                # Anzahl Bedingungen
```

```
# Faktor Gruppenzugehörigkeit
> IV <- factor(rep(1:P, each=nBl),
+                 labels=c("Caffeine", "Single", "Double", "Placebo"))

> blocks <- factor(rep(1:nBl, times=P))      # Faktor Blockzugehörigkeit
> fDf    <- data.frame(IV, DV, blocks)        # Datensatz
> friedman.test(DV ~ IV | blocks, data=fDf)
Friedman rank sum test
data: DV and IV and blocks
Friedman chi-squared = 8.2653, df = 3, p-value = 0.04084
```

Die Ausgabe nennt den Wert der asymptotisch χ^2 -verteilten Teststatistik gefolgt von den Freiheitsgraden (df) und dem p-Wert (p-value). Das Ergebnis lässt sich auch manuell nachvollziehen, wobei das zentrale Element der Teststatistik das Quadrat der pro Gruppe gebildeten Summe der Ränge innerhalb jedes Blocks ist.²⁰

```
> (DVmat <- cbind(DVcaff, DVds, DVdd, DVplac))      # Datenmatrix
   DVcaff  DVds  DVdd  DVplac
[1,]    14    11    16    13
[2,]    13    12    15    12
[3,]    12    13    14    11
[4,]    11    14    13    10
[5,]    10    15    12     9

> (rankMat <- t(apply(DVmat, 1, rank)))      # Matrix blockweise Ränge
   DVcaff  DVds  DVdd  DVplac
[1,]    3    1.0    4    2.0
[2,]    3    1.5    4    1.5
[3,]    2    3.0    4    1.0
[4,]    2    4.0    3    1.0
[5,]    2    4.0    3    1.0

> (rankSumJ <- colSums(rankMat))            # gruppenweise Rangsummen
   DVcaff  DVds  DVdd  DVplac
12.0   13.5  18.0    6.5

# Teststatistik
> (S <- (12 / (nBl*P*(P+1))) * sum(rankSumJ^2) - 3*nBl*(P+1))
[1] 8.1

> (pVal <- pchisq(S, P-1, lower.tail=FALSE))      # p-Wert
[1] 0.04398959
```

Für den Page-Trend-Test für geordnete abhängige Gruppen vgl. `PageTest()` aus dem Paket `DescTools`.

²⁰Im gewählten Beispiel sind die Ränge im zweiten Block nicht eindeutig, es treten also Bindungen auf. Deshalb wird die Teststatistik S in `friedman.test()` weiter korrigiert und stimmt nicht exakt mit der hier berechneten überein.

10.5.8 Cochran-Q-Test für abhängige Stichproben

Cochrancs *Q*-Test ist analog zum Friedman-Test für den Fall, dass die AV dichotom ist. Bei diesem Test werden die Werte der AV allerdings nicht zunächst in Ränge umgewandelt, sondern bei der Codierung mit 0 und 1 belassen. Es liegen abhängige Daten aus p Bedingungen vor: Entweder liefert jedes Beobachtungsobjekt Werte aus jeder Bedingung (Messwiederholung), oder aber p homogene (gematchte) Beobachtungsobjekte werden zu einem Block zusammengefasst, von dem dann p abhängige Werte aus den Bedingungen stammen. Der *Q*-Test wird mit `symmetry_test()` aus dem Paket `coin` durchgeführt und prüft die H_0 , dass die Trefferwahrscheinlichkeit in allen Bedingungen identisch ist.

```
> symmetry_test(formula=<Modellformel>, data=<Datensatz>, subset=NULL,
+                 teststat="quad")
```

Die Daten müssen im Long-Format vorliegen (vgl. Abschn. 3.3.9). Unter `formula` ist eine Modellformel der Form $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle + \langle \text{BlockId} \rangle$ zu nennen, wobei $\langle \text{UV} \rangle$ ein Faktor derselben Länge wie $\langle \text{AV} \rangle$ ist und für jede Beobachtung in $\langle \text{AV} \rangle$ die zugehörige UV-Stufe angibt. $\langle \text{BlockId} \rangle$ ist ebenfalls ein Faktor derselben Länge wie $\langle \text{AV} \rangle$ und enthält die Blockzugehörigkeit jedes Wertes. Stammen die verwendeten Variablen aus einem Datensatz, muss dieser unter `data` eingetragen werden. Das Argument `subset` erlaubt es, nur eine Teilmenge der Fälle einfließen zu lassen, es erwartet einen entsprechenden Indexvektor, der sich auf die Zeilen des Datensatzes bezieht. Für den *Q*-Test ist zudem das Argument `teststat="quad"` zu setzen, da `symmetry_test()` es erlaubt, verschiedene Testverfahren für dieselbe Hypothese anzuwenden.

Als Beispiel diene jenes aus Büning und Trenkler (1994, p. 208 ff.): Über fünf Jahre hinweg geben dieselben zehn Wahlberechtigten als dichotomes Präferenzurteil an, ob sie eine bestimmte Partei den anderen vorziehen.

```
# Präferenzurteile der 10 Blöcke in den 5 Jahren
> pref <- c(1,1,0,1,0, 0,1,0,0,1, 1,0,1,0,0, 1,1,1,1,1, 0,1,0,0,0,
+           1,0,1,1,1, 0,0,0,0,0, 1,1,1,1,0, 0,1,0,1,1, 1,0,1,0,0)

> N      <- 10                                # Anzahl Blöcke / VPn
> year   <- factor(rep(1981:1985, times=N))  # Faktor Messzeitpunkt
> P      <- nlevels(year)                      # Anzahl Messzeitpunkte
> id     <- factor(rep(1:N, each=P))          # Faktor Blockzugehörigkeit
> library(coin)                               # für symmetry_test()
> symmetry_test(pref ~ year | id, teststat="quad")
Asymptotic General Independence Test
data: pref by year (1981, 1982, 1983, 1984, 1985)
stratified by id
chi-squared = 1.3333, df = 4, p-value = 0.8557
```

Die Ausgabe nennt den Wert der asymptotisch χ^2 -verteilten Teststatistik gefolgt von den Freiheitsgraden (`df`) und dem *p*-Wert (`p-value`). Das Ergebnis lässt sich auch manuell nachvollziehen, wobei die Messwerte zunächst als Datenmatrix im Wide-Format zusammenzufassen sind. Das zentrale Element der Teststatistik sind die Abweichungen der Zeilen- und Spaltensummen in dieser Matrix von ihrem jeweiligen Mittel.

```
# Datenmatrix im Wide-Format
> prefMat <- matrix(pref, nrow=N, ncol=P, byrow=TRUE)
> rSum      <- rowSums(prefMat)                      # Zeilensummen
> cSum      <- colSums(prefMat)                     # Spaltensummen

# Teststatistik Q
> (Q <- (P*(P-1)*sum((cSum-mean(cSum))^2)) / (P*sum(rSum) - sum(rSum^2)))
[1] 1.333333

> (pVal <- pchisq(Q, P-1, lower.tail=FALSE))       # p-Wert
[1] 0.8556952
```

10.5.9 Bowker-Test für zwei abhängige Stichproben

Der Bowker-Test ist für Situationen geeignet, in denen eine kategoriale Variable mit mehr als zwei Ausprägungen in zwei abhängigen Stichproben beobachtet wird. Analog zum Friedman-Test besteht die H_0 darin, dass die Verteilung der Variable in beiden Bedingungen identisch ist und damit eine bzgl. der Hauptdiagonale symmetrische Kontingenztafel der gemeinsamen Häufigkeiten vorliegt. Die ungerichtete H_1 besagt, dass es eine systematische Abweichung von Übereinstimmung in eine Richtung gibt. `mcnemar.test()` berechnet automatisch den Bowker-Test, wenn eine Kontingenztafel mit jeweils mehr als zwei Zeilen und Spalten als Argument übergeben wird.

```
> mcnemar.test(x, y=NULL)
```

Unter `x` kann die quadratische Kontingenztafel der beiden Datenvektoren einer kategorialen Variable aus zwei abhängigen Stichproben angegeben werden. Pro Beobachtungseinheit übereinstimmende Ausprägungen der Variable stehen in dieser Matrix in der Diagonale, voneinander abweichende Ausprägungen außerhalb der Diagonale. Wird für `x` stattdessen ein die Daten aus einer Stichprobe codierendes Objekt der Klasse `factor` genannt, muss auch `y` ein Faktor mit denselben Stufen und derselben Länge wie `x` sein, der die Daten der anderen Stichprobe speichert.

Als Beispiel diene jenes aus Bortz et al. (2010, p. 165 ff.): An denselben Personen soll die empfundene Leistungssteigerung als Wirkung eines Medikaments oder Placebos untersucht werden. Erhoben wird die Einschätzung, ob keine, eine geringe, oder eine starke Wirkung vorliegt.

```
> categ <- factor(1:3, labels=c("lo", "med", "hi"))    # AV-Kategorien
> Q      <- nlevels(categ)                            # Anzahl Kategorien
> drug   <- rep(categ, c(30, 50, 20))              # Daten Medikament

# Daten Placebo
> plac   <- rep(rep(categ, length(categ)), c(14,7,9, 5,26,19, 1,7,12))
> cTabBow <- xtabs(~ drug + plac)                  # Kontingenztafel
> addmargins(cTabBow)                                # Randsummen
> plac
```

drug	lo	med	hi	Sum
lo	14	7	9	30
med	5	26	19	50
hi	1	7	12	20
Sum	20	40	40	100

```
> mcnemar.test(cTabBow)
McNemar's Chi-squared test
data: cTabBow
McNemar's chi-squared = 12.2718, df = 3, p-value = 0.006508
```

Die Ausgabe nennt den Wert der asymptotisch χ^2 -verteilten Teststatistik gefolgt von den Freiheitsgraden (df) und dem p-Wert (p-value). Da die Symmetrie einer Kontingenztafel zu testen ist, eignet sich hier wie bei Cochran's Q-Test auch die Funktion `symmetry_test()` aus dem `coin` Paket zur Auswertung. Sie akzeptiert die Kontingenztafel der Übereinstimmungen beider Bedingungen als Argument.

```
> library(coin) # für symmetry_test()
> symmetry_test(cTab, teststat="quad") # ...
```

Teststatistik des Bowker-Tests ist die Summe der quadrierten Differenzen von an der Hauptdiagonale gespiegelten Einträgen der Kontingenztafel, die zuvor an der Summe beider Einträge relativiert wurden.

```
# relativierte quadrierte Differenzen Kontingenztafel vs. Transponierte
> sqDiffs <- (cTabBow - t(cTabBow))^2 / (cTabBow + t(cTabBow))

# summiere nur über die Differenzen der oberen Dreiecksmatrix
> (chisqVal <- sum(sqDiffs[upper.tri(cTabBow)])) # Teststatistik
[1] 12.27179

> (bowDf <- choose(Q, 2)) # Freiheitsgrade Q*(Q-1)/2
[1] 3

> (pVal <- pchisq(chisqVal, bowDf, lower.tail=FALSE)) # p-Wert
[1] 0.006507799
```

10.5.10 McNemar-Test für zwei abhängige Stichproben

Ein Spezialfall des Bowker-Tests ist der McNemar-Test für Daten einer dichotomen Variable aus zwei abhängigen Stichproben. Seine H_0 , dass die Verteilung der AV in beiden Bedingungen identisch ist, lässt sich auch so formulieren, dass die Kontingenztafel der Übereinstimmungen der Daten aus den abhängigen Stichproben bzgl. der Hauptdiagonale symmetrisch ist. Die ungerichtete H_1 besagt, dass es eine systematische Abweichung von Übereinstimmung in eine Richtung gibt. Wie im Bowker-Test kann die Hypothese mit `mcnemar.test()` geprüft werden. Zusätzlich legt hier das Argument `correct` fest, ob eine Stetigkeitskorrektur durchgeführt wird (Voreingestellung: TRUE).

Im Beispiel sei an einer Stichprobe jeweils vor und nach einer Informationskampagne die Variable erhoben worden, ob eine Person raucht.

```
> N      <- 20                                # Anzahl Versuchspersonen
> pre   <- rbinom(N, size=1, prob=0.6)        # Prä-Messung
> post  <- rbinom(N, size=1, prob=0.4)        # Post-Messung

# Konvertierung der numerischen Vektoren in Faktoren
> preFac <- factor(pre,    labels=c("no", "yes"))
> postFac <- factor(post,   labels=c("no", "yes"))
> P       <- nlevels(preFac)                  # Anzahl Kategorien
> cTab   <- xtabs(~ preFac + postFac)        # Kontingenztafel
> addmargins(cTab)                           # Randsummen
      postFac
preFac no yes Sum
  no    4   6 10
  yes   5   5 10
  Sum   9  11 20

> mcnemar.test(cTab, correct=FALSE)
McNemar's Chi-squared test
data: cTab
McNemar's chi-squared = 0.0909, df = 1, p-value = 0.763
```

Wie beim Bowker-Test eignet sich auch hier die Funktion `symmetry_test()` aus dem `coin` Paket zur Auswertung, die als Argument die Kontingenztafel der Übereinstimmungen beider Bedingungen erwartet.

```
> library(coin)                               # für symmetry_test()
> symmetry_test(cTab, teststat="quad")        # ...
```

Verglichen mit dem Bowker-Test vereinfacht sich bei der manuellen Berechnung die Formel für die Teststatistik, da in der Kontingenztafel nun nur noch die Differenz der beiden Zellen außerhalb der Diagonale zu berücksichtigen ist.

```
# Teststatistik
> (chisqVal <- (cTab[1, 2] - cTab[2, 1])^2 / (cTab[1, 2] + cTab[2, 1]))
[1] 0.0909091

> (mcnDf <- choose(P, 2))                  # Freiheitsgrade P*(P-1)/2
[1] 1

> (pVal <- pchisq(chisqVal, mcnDf, lower.tail=FALSE))      # p-Wert
[1] 0.7630246
```

Treten kleine erwartete Zellhäufigkeiten auf (etwa < 5), kann die χ^2 -Approximation der Verteilung der Teststatistik noch ungenau sein. In diesem Fall lässt sich ein *exakter* McNemar-Test mit Hilfe eines ungerichteten Binomialtests durchführen: Er wertet eine Zelle außerhalb der Diagonale der Kontingenztafel als Anzahl der Treffer, die andere Zelle außerhalb der Diagonale

als Anzahl der Nicht-Treffer und testet gegen die $H_0: p_0 = 0.5$. Dieser Test gilt jedoch als recht konservativ.

```
# Zellen der Gegendiagonale als Anzahl der Treffer, Nicht-Treffer
> binom.test(c(cTab[1, 2], cTab[2, 1]), p=0.5)
Exact binomial test
data: c(cTab[1, 2], cTab[2, 1])
number of successes = 6, number of trials = 11, p-value = 1
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.2337936 0.8325119
sample estimates:
probability of success
 0.5454545
```

10.5.11 Stuart-Maxwell-Test für zwei abhängige Stichproben

Der Stuart-Maxwell-Test prüft die Kontingenztafel zweier kategorialer Variablen auf Homogenität der Randverteilungen. Der Test kann z. B. zur Beurteilung der Frage eingesetzt werden, ob zwei rater die verfügbaren Kategorien mit denselben Grundwahrscheinlichkeiten verwenden (vgl. Abschn. 10.3.3). Verglichen mit dem Bowker-Test bezieht er sich auf nur einen Spezialfall, der zu einer asymmetrischen Kontingenztafel der Übereinstimmungen führen kann.

Zur Berechnung des Tests steht aus dem `coin` Paket die Funktion `mh_test()` bereit. Sie erwartet als Argument die Kontingenztafel der Übereinstimmungen der kategorialen AV in beiden Bedingungen. Als Beispiel sei wie beim Bowker-Test jenes aus Bortz et al. (2010, p. 165 ff.) herangezogen.

```
> library(coin)                      # für mh_test()
> mh_test(cTabBow)
Asymptotic Marginal-Homogeneity Test
data: response by groups (drug, plac) stratified by block
chi-squared = 12.1387, df = 2, p-value = 0.002313
```

Die manuelle Prüfung ist für den Fall von (3×3) -Kontingenztafeln wie folgt möglich:

```
> addmargins(cTabBow)                # Kontingenztafel mit Randhäufigkeiten
                                         plac
   drug lo med hi Sum
   lo    14    7   9  30
   med   5   26  19  50
   hi    1    7  12  20
   Sum   20   40  40 100

# Hälfte der Summe der an der Hauptdiagonale gespiegelten Häufigkeiten
> (Nij <- ((cTabBow + t(cTabBow)) / 2)[upper.tri(cTabBow)])
[1] 6 5 13
```

```
# Abweichungen der Randverteilungen
> (d <- rowSums(cTabBow) - colSums(cTabBow))
lo med hi
10 10 -20

> num <- sum(Nij * rev(d^2)) # Zähler
> denom <- 2 * sum(apply(combn(Nij, 2), 2, prod)) # Nenner
> (chisqVal <- num / denom) # Teststatistik
[1] 12.13873

> (smmhDf <- nrow(cTabBow)-1) # Freiheitsgrade
[1] 2

> (pVal <- pchisq(chisqVal, smmhDf, lower.tail=FALSE)) # p-Wert
[1] 0.002312643
```

Im allgemeinen Fall ist es zur Bestimmung der Teststatistik notwendig, die Kovarianzmatrix der Abweichungen beider Randverteilungen unter H_0 zu bestimmen und zu invertieren (vgl. Abschn. 12.1.2). Besitzt die Variable p Kategorien, reicht bereits die Information über die Abweichung in den Randverteilungen bzgl. der ersten $p - 1$ Kategorien aus, da sich die Randsummen der Kontingenztafel zur Anzahl der Objekte summieren.

```
# spätere Kovarianzmatrix der Abweichungen der Randverteilungen unter H0
> S <- -(cTabBow + t(cTabBow))

# setze Diagonale dieser Kovarianzmatrix
> diag(S) <- rowSums(cTabBow) + colSums(cTabBow) - 2*diag(cTabBow)

# berücksichtige für Teststatistik nur die ersten P-1 Kategorien
> keep <- 1:(nrow(cTabBow)-1)
> (chisqVal <- t(d[keep]) %*% solve(S[keep, keep]) %*% d[keep])
[1] 12.13873
```

Kapitel 11

Resampling-Verfahren

Resampling-Verfahren kommen für eine Vielzahl von Tests in Frage, können hier aber nur in Grundzügen vorgestellt werden. Ausgangspunkt ist die gesuchte Verteilung einer Teststatistik $\hat{\theta}$ – etwa eines Schätzers $\hat{\theta}$ für einen theoretischen Parameter θ . Diese Verteilung kann aus verschiedenen Gründen unbekannt sein: So sind etwa die in parametrischen Tests gemachten Annahmen, unter denen ihre Teststatistik eine bekannte Verteilung aufweist, nicht immer zu rechtfertigen. In vielen klassischen nonparametrischen Verfahren ist die Verteilung der Teststatistik zwar im Prinzip exakt zu ermitteln, praktisch aber der Rechenaufwand dafür zu hoch.

Grundidee von Bootstrap-Verfahren und Permutationstests ist es, aus den gegebenen Daten einer festen Basisstichprobe viele neue Zufallsstichproben (*resamples*) zu generieren und die Teststatistik für jedes resample zu ermitteln – die dabei berechneten Werte werden als $\hat{\theta}^*$ bezeichnet. Die empirische Verteilung der $\hat{\theta}^*$ dient der Approximation der theoretischen Verteilung von $\hat{\theta}$. Die Beziehung zwischen resample und Basisstichprobe wird also auf die Beziehung zwischen Basisstichprobe und Population übertragen. Im Vergleich zu klassischen nonparametrischen Tests (vgl. Kap. 10) versprechen Resampling-Methoden oft eine höhere Power.

11.1 Bootstrapping

Bei Bootstrap-Verfahren (Chernik & LaBudde, 2011; Chihara & Hesterberg, 2011; Davison & Hinkley, 1997) werden die resamples als *Replikationen* bezeichnet. Dies sind mit Zurücklegen gezogene Zufallsstichproben aus der Basisstichprobe mit dem Stichprobenumfang. $\hat{\theta}$ ist der auf Grundlage der Basisstichprobe berechnete *plug-in*-Schätzer von θ , wird also auf empirischer Ebene rechnerisch genauso gebildet wie θ auf theoretischer Ebene.¹ Über den – oft weniger interessanten – Bootstrap-Punktschätzer für θ hinaus lassen sich aus der empirischen Verteilung von $\hat{\theta}^*$ vor allem Konfidenzintervalle für θ bestimmen. Dafür approximiert die empirische Verteilung von $\hat{\theta}^* - \hat{\theta}$ in vielen Fällen jene der Pivot-Statistik $\hat{\theta} - \theta$, deren Verteilung unabhängig vom konkreten Wert für θ ist.

Ein vollständiger Bootstrap umfasst alle möglichen Replikationen einer Basisstichprobe vom Umfang n , wofür der Rechenaufwand jedoch meist zu groß ist: Es gibt bereits $2^n - 1$ relevante Teilmengen von Beobachtungsobjekten (die Mächtigkeit der Potenzmenge ohne die leere Menge),

¹ θ ist ein *Funktional* der theoretischen Verteilungsfunktion F der ursprünglichen Zufallsvariable, bildet also F auf eine Zahl ab. Analog ist $\hat{\theta}$ dasselbe Funktional der empirischen kumulativen Häufigkeitsverteilung \hat{F}_n der Basisstichprobe vom Umfang n und $\hat{\theta}^*$ dasselbe Funktional der empirischen kumulativen Häufigkeitsverteilung \hat{F}_n^* in einer Replikation.

wobei die Elemente jeder denkbaren Zusammensetzung noch mit unterschiedlichen Häufigkeiten berücksichtigt werden können. Deswegen wird bootstrapping als *Monte-Carlo-Verfahren* durchgeführt und nur eine zufällige Auswahl von Replikationen berücksichtigt.

11.1.1 Replikationen erstellen

Das Paket `boot` (Canty & Ripley, 2014) stellt mit `boot()` eine Funktion bereit, die Bootstrap-Replikationen durchführt.² Hier wird nur ihre Verwendung für das nonparametrische bootstrapping beschrieben.

```
> boot(data=Basisstichprobe, statistic=Funktion, R=# Replikationen),
+       strata=Faktor)
```

Für `data` ist der Vektor mit den Werten der Basisstichprobe zu übergeben. Basiert $\hat{\theta}$ auf Daten mehrerer Variablen, muss `data` ein Datensatz mit diesen Variablen sein. Das Argument `statistic` erwartet den Namen einer Funktion mit ihrerseits zwei Argumenten zur Berechnung von $\hat{\theta}^*$: Ihr erstes Argument ist ebenfalls der Vektor bzw. Datensatz der Basisstichprobe. Das zweite Argument von `statistic` ist ein Indexvektor, dessen Elemente sich auf die beobachteten Fälle beziehen. `boot()` ruft für jede der `R` vielen Replikationen `statistic` auf und übergibt die Basis-Daten samt eines zufällig gewählten Indexvektors als Anweisung, wie eine konkrete Replikation aus Fällen der Basisstichprobe zusammengesetzt sein soll. Das Ergebnis von `statistic` muss $\hat{\theta}^*$ sein – sind gleichzeitig mehrere Parameter θ_j zu schätzen, analog ein Vektor mit den $\hat{\theta}_j^*$. Um beim bootstrapping eine vorgegebene Stratifizierung der Stichprobe beizubehalten, kann ein Faktor an `strata` übergeben werden, der eine Gruppeneinteilung definiert. In den Replikationen sind die Gruppengrößen dann gleich jenen der Basisstichprobe.

Als Beispiel diene die die Situation eines *t*-Tests für eine Stichprobe auf einen festen Erwartungswert μ_0 (vgl. Abschn. 7.2.1). Ziel im folgenden Abschnitt ist die Konstruktion eines Vertrauensintervalls für μ ($= \theta_1$). Die Stichprobengröße sei n , der Mittelwert M ($= \hat{\theta}_1$) und die unkorrigierte Varianz des Mittelwertes S_M^2 ($= \hat{\sigma}_{\hat{\theta}_1}^2 = \hat{\theta}_2$) als plug-in-Schätzer der theoretischen Varianz σ_M^2 ($= \theta_2$). Dazu ist aus jeder Bootstrap-Replikation der Mittelwert M^* ($= \hat{\theta}_1^*$) und die unkorrigierte Varianz des Mittelwertes S_M^{2*} ($= \hat{\sigma}_{\hat{\theta}_1^*}^2 = \hat{\theta}_2^*$) zu berechnen. Für das Erstellen eigener Funktionen vgl. Abschn. 15.2.

```
> muH0 <- 100                                # Erwartungswert unter H0
> sdH0 <- 40                                  # Streuung unter H0
> N      <- 200                                # Größe Basisstichprobe
> DV    <- rnorm(N, muH0, sdH0)                # Basisstichprobe

# Mittelwert M* und Varianz des Mittelwertes S(M)^2* aus BS-Replikation,
# deren zufällige Zusammensetzung durch Indexvektor idx bestimmt wird
> getM <- function(orgDV, idx) {
+   n      <- length(orgDV[idx])
+   bsM   <- mean(orgDV[idx])                   # M*
+   bsS2M <- (((n-1)/n) * var(orgDV[idx])) / n  # S(M)^2*
```

²Flexible bootstrap-basierte Tests für eine oder zwei Stichproben ermöglicht auch das Paket `resample` (Hesterberg, 2014).

```

+      c(bsM, bsS2M)
}

> library(boot)                      # für boot(), boot.ci()
> nR      <- 999                     # Anzahl BS-Replikationen
> (bsRes <- boot(DV, statistic=getM, R=nR))      # BS-Replikationen
ORDINARY NONPARAMETRIC BOOTSTRAP
Call:
boot(data = DV, statistic = getM, R = nR)

Bootstrap Statistics :
      original       bias   std. error
t1*  102.655867 -0.09535249   3.057508
t2*   9.162478  -0.02025724   0.881103

```

Die Ausgabe von `boot()` nennt in den Zeilen `t1*` und `t2*` Eigenschaften der von `statistic` zurückgegebenen Bootstrap-Schätzer: In der Spalte `original` stehen die für die Basisstichprobe berechneten Werte. Hier sind dies der Mittelwert und seine unkorrigierte Varianz. In der Spalte `bias` folgt die Verzerrung jedes Bootstrap-Schätzers als Mittelwert der Abweichungen $\hat{\theta}^* - \hat{\theta}$. Für die Punktschätzung von θ kann sie zu einer einfachen Bias-Korrektur eingesetzt werden, indem sie von $\hat{\theta}$ abgezogen wird. Schließlich folgt in der Spalte `std. error` die korrigierte Streuung für jeden Kennwert der pro Replikation von `statistic` berechneten Kennwerte. Diese sind in der von `boot()` zurückgegebenen Liste spaltenweise als Matrix in der Komponente `t` gespeichert.

```

> (M <- mean(DV))                  # Mittelwert Basisstichprobe
[1] 102.6559

> (S2M <- (((N-1)/N) * var(DV)) / N)      # unkorrig. Varianz von M
[1] 9.162478

> Mstar    <- bsRes$t[, 1]                # M* jeder Replikation
> S2Mstar <- bsRes$t[, 2]                # S(M)^2* jeder Replikation
> (biasM  <- mean(Mstar) - M)           # Bias-Schätzung für M
[1] -0.0953525

> mean(S2Mstar) - S2M                 # Bias-Schätzung für S(M)^2
[1] -0.02025724

> c(sd(Mstar), sd(S2Mstar))          # Streuungen der BS-Kennwerte
[1] 3.057508 0.881103

```

Die Verteilung von $\hat{\theta}^* - \hat{\theta}$ sollte unimodal und symmetrisch sein, nicht unähnlich einer Normalverteilung (Abb. 11.1). Dies lässt sich etwa durch ein Histogramm mit eingezzeichneter Dichtefunktion einer passenden Normalverteilung zusammen mit einem nonparametrischen Kerndichteschätzer oder einem Q-Q-Plot prüfen.

```
> hist(Mstar-M, freq=FALSE, breaks="FD") # Histogramm von M* - M
```

```
> rug(jitter(Mstar-M)) # Einzelwerte der M* - M

# Dichtefunktion einer Normalverteilung
> curve(dnorm(x, mean(Mstar-M), sd(Mstar-M)), lwd=2, col="blue", add=TRUE)
> lines(density(Mstar-M), lwd=2, col="red", lty=2) # Kerndichteschätzer
> qqnorm(Mstar-M, pch=16) # Q-Q-Plot der M* - M
> qqline(Mstar-M, lwd=2, col="blue") # Normalverteilung-Referenz
```

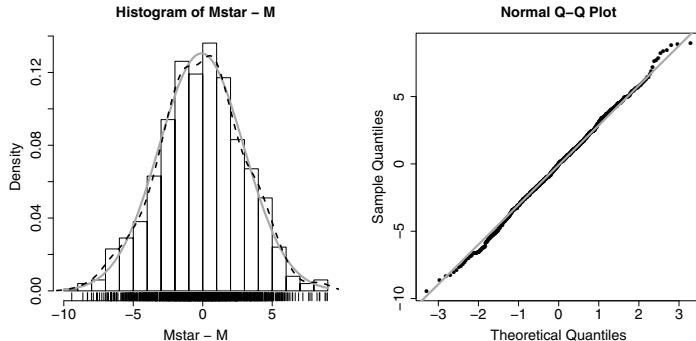


Abbildung 11.1: Histogramm von $M^* - M$ aus Bootstrap-Replikationen mit passender Normalverteilung und nonparametrischem Kerndichteschätzer. Q-Q-Plot von $M^* - M$ mit Vergleich zur Standardnormalverteilung.

Detaillierte Informationen über die Zusammensetzung aller von `boot()` erstellten Replikationen liefert `boot.array(boot-Objekt, indices=TRUE)`. Bei einer Basisstichprobe vom Umfang n und R Replikationen ist das Ergebnis mit dem Argument `indices=TRUE` eine $(R \times n)$ -Matrix mit einer Zeile für jede Replikation und einer Spalte pro Beobachtung. Die Zelle (r, i) enthält den Index des ausgewählten Elements der Basisstichprobe. Zusammen mit dem Vektor der Basisstichprobe lassen sich damit alle Replikation rekonstruieren.

```
> bootIdx <- boot.array(bsRes, indices=TRUE)
# Replikationen 1-3: je ausgewählte erste 10 Indizes Basisstichprobe
> bootIdx[1:3, 1:10]
[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,] 78 148 83 72 86 78 33 59 87 141
[2,] 41 116 197 87 183 4 102 129 84 175
[3,] 123 107 60 181 35 133 196 197 15 71

> repl1Idx <- bootIdx[1, ] # Indizes der ersten Replikation
> repl1DV <- DV[repl1Idx] # Werte der ersten Replikation
> head(repl1DV, n=5)
[1] 86.12395 72.41568 135.87073 149.96011 99.03936
```

11.1.2 Bootstrap-Vertrauensintervalle für μ

Ein von `boot()` erzeugtes Objekt ist für das Argument `boot.out` der Funktion `boot.ci()` anzugeben, die das zweiseitige Konfidenzintervall für θ bestimmt.

```
> boot.ci(boot.out=<boot-Objekt>, conf=<Breite VI>, index=<Nummer>,
+           type="<Methode>")
```

Gibt die für `boot(..., statistic)` genannte Funktion bei jedem Aufruf J geschätzte Parameter zurück, ist `index` der Reihe nach auf $1, \dots, J$ zu setzen, um das zugehörige Konfidenzintervall zu erhalten. Das Intervall wird mit der Breite `conf` nach einer über `type` festzulegenden Methode gebildet:

- "`basic`": Das klassische Bootstrap-Intervall um $\hat{\theta}$, dessen Breite durch die $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantile der Werte von $\hat{\theta} - \hat{\theta}^*$ definiert ist.
- "`perc`": Das Perzentil-Intervall, dessen Grenzen die $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantile der Werte von $\hat{\theta}^*$ sind.
- "`norm`": Das Normalverteilungs-Intervall mit dem Zentrum in $\hat{\theta} - M(\hat{\theta}^* - \hat{\theta})$ (Bias-Korrektur), dessen Breite definiert ist durch die $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantile der Standardnormalverteilung multipliziert mit der Streuung von $\hat{\theta}^*$.
- "`stud`": Das t -Vertrauensintervall um $\hat{\theta}$, dessen Breite definiert ist durch die $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantile der Werte von $t^* = \frac{\hat{\theta}^* - \hat{\theta}}{S_{\hat{\theta}}^*}$ multipliziert mit der Streuung von $\hat{\theta}^*$. Voraussetzung ist, dass die Funktion `statistic` als zweites Element den plug-in Schätzer $S_{\hat{\theta}}^{2*}$ der theoretischen Varianz $\sigma_{\hat{\theta}}^2$ zurückliefert. Ist deren geschlossene Form unbekannt oder existiert nicht, lässt sich $S_{\hat{\theta}}^{2*}$ innerhalb jedes Aufrufs von `statistic` im ursprünglichen bootstrapping durch eine eigene (*nested*) Bootstrap-Schätzung ermitteln.
- "`bca`": Das BC_a -Intervall (*bias-corrected and accelerated*). Für symmetrische Verteilungen und Schätzer mit geringem bias ähnelt es dem Perzentil- und t -Intervall meist stark. Bei schießen Verteilungen und größerem bias wird das BC_a -Intervall den anderen oft vorgezogen.

Bei den Intervallen gehen zur Erhöhung der Genauigkeit nicht die $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantile selbst von t^* bzw. von $\hat{\theta}^*$ ein, die mit `quantile(..., probs=c(<alpha>/2, 1 - <alpha>/2))` zu ermitteln wären: Bei n_R vielen Replikationen sind dies stattdessen die Elemente mit den Indizes $(n_R + 1) \cdot \frac{\alpha}{2}$ und $(n_R + 1) \cdot (1 - \frac{\alpha}{2})$ der sortierten Werte von t^* bzw. von $\hat{\theta}^*$. Ergibt sich kein ganzzahliges Ergebnis für die Indizes, interpoliert `boot.ci()` jeweils zwischen den angrenzenden Elementen.

```
> alpha <- 0.05                                     # alpha-Niveau
> boot.ci(bsRes, conf=1-alpha,
+           type=c("basic", "perc", "norm", "stud", "bca"))
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 999 bootstrap replicates
CALL :
boot.ci(boot.out=bsRes, conf=1 - alpha,
```

```

type = c("basic", "perc", "norm", "stud", "bca"))

Intervals :
Level      Normal          Basic       Studentized
95%  ( 96.8, 108.7 )  ( 97.0, 109.1 )  ( 96.7, 109.0 )

Level      Percentile        BCa
95%  ( 96.2, 108.3 )  ( 96.2, 108.3 )
Calculations and Intervals on Original Scale

# Kontrolle: Indizes (n_R + 1) * alpha/2 und (n_R + 1) * (1-alpha/2)
> (idx <- trunc((nR + 1) * c(alpha/2, 1 - alpha/2)))
[1] 25 975

> tStar <- (Mstar-M) / sqrt(S2Mstar)           # t*
> tCrit <- sort(tStar)[idx]                     # 2.5%, 97.5%-Quantil von t*
> zCrit <- qnorm(c(alpha/2, 1 - alpha/2))      # 2.5%, 97.5%-Quantil N(0, 1)
> (ciBasic <- 2*M - sort(Mstar)[idx])          # klassisches VI
[1] 109.11192 96.97443

> (ciPerc <- sort(Mstar)[idx])                 # Perzentil-VI
[1] 96.19981 108.33731

> (ciNorm <- M-biasM - zCrit*sd(Mstar))       # N(0, 1)-VI
[1] 108.74383 96.75861

> (ciT <- M - tCrit*sqrt(S2M))                # t-VI
[1] 108.99497 96.70034

```

Bei der manuellen Umsetzung der Bootstrap-Schätzungen sollen als Maß für deren Güte die kumulierten relativen Häufigkeiten von $t^* = \frac{M^* - M}{S_M^*}$ mit der Verteilungsfunktion von $t = \frac{M - \mu_0}{s/\sqrt{n}}$ verglichen werden (mit s als korrigierter Streuung). Im Fall n unabhängiger Realisierungen einer normalverteilten Variable ist dies die t_{n-1} Verteilung (Abb. 11.2).

```

# M*, S(M)* und t* aus Bootstrap-Replikationen
> res    <- replicate(nR, getM(DV, sample(seq(along=DV), replace=TRUE)))
> Mstar  <- res[1, ]                                # M*
> SMstar <- sqrt(res[2, ])                          # S(M)*
> tStar   <- (Mstar-mean(DV)) / SMstar            # t*

# kumulierte relative Häufigkeiten von t*
> plot(tStar, ecdf(tStar)(tStar), col="gray60", pch=1,
+       xlab="t* bzw. t", ylab="P(T <= t)",
+       main="t*: Kumulierte rel. Häufigkeiten und Verteilungsfunktion")

# theoretische Verteilungsfunktion von t und Legende
> curve(pt(x, N-1), lwd=2, add=TRUE)

```

```
> legend(x="topleft", lty=c(NA, 1), pch=c(1, NA),
+         lwd=c(2, 2), col=c("gray60", "black"), legend=c("t*", "t"))
```

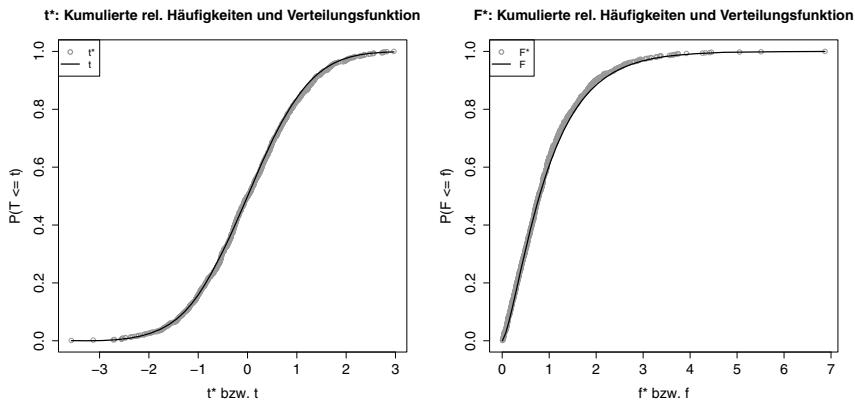


Abbildung 11.2: Kumulierte relative Häufigkeiten von t^* aus Bootstrap-Replikationen mit theoretischer Verteilungsfunktion t_{n-1} . Kumulierte relative Häufigkeiten von F^* aus Bootstrap-Replikationen mit theoretischer Verteilungsfunktion $F_{p-1, n-p}$.

11.1.3 Bootstrap-Vertrauensintervalle für $\mu_2 - \mu_1$

In der Situation eines t -Tests für zwei unabhängige Stichproben (vgl. Abschn. 7.2.2) kann bootstrapping eine nonparametrische Schätzung des Konfidenzintervalls für die Differenz der Erwartungswerte $\theta = \mu_2 - \mu_1$ liefern. In der Basisstichprobe ist die Differenz der Gruppenmittelwerte $\hat{\theta} = M_2 - M_1$ ein Schätzer für θ , in jeder Replikation analog $\hat{\theta}^* = M_2^* - M_1^*$. Für die Replikationen ist zu beachten, dass jeweils innerhalb jeder Gruppe mit Zurücklegen aus der Basisstichprobe gezogen wird, damit Gruppenzugehörigkeit und Gruppengrößen erhalten bleiben. Dies lässt sich mit dem Argument `strata` von `boot()` erreichen. Als Beispiel diene jenes aus Abschn. 7.2.2 mit einer bei Frauen und Männern erhobenen Variable.

```
# Datensatz aus Variable bei Männern, bei Frauen und Gruppierungsfaktor
> tDf <- data.frame(DV=c(DVm, DVf), # Variable bei Männern und Frauen
+                      IV=factor(rep(c("m", "f"), c(n1, n2)))) 

# Funktion, um Differenz der Mittelwerte von f und m zu berechnen
> getDM <- function(dat, idx) {
+   # Gruppenmittelwerte für durch idx definierte Beobachtungen
+   Mfm <- aggregate(DV ~ IV, data=dat, subset=idx, FUN=mean)
+   -diff(Mfm$DV)      # M-Differenz, Reihenfolge f-m wie in t.test()
+ }

> library(boot)                      # für boot(), boot.ci()
> bsTind <- boot(tDf, statistic=getDM, strata=tDf$IV, R=999)
> boot.ci(bsTind, conf=0.95, type=c("basic", "bca"))
```

```
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 999 bootstrap replicates
CALL :
boot.ci(boot.out = bsTind, conf = 0.95, type = c("basic", "bca"))

Intervals :
Level      Basic              BCa
95%  (-8.157,  0.878 )  (-8.529,  0.532 )
Calculations and Intervals on Original Scale
```

Als Vergleich diene das parametrische Konfidenzintervall aus dem zugehörigen t -Test.

```
> tt <- t.test(DV ~ IV, alternative="two.sided", var.equal=TRUE, data=tDf)
> tt$conf.int
[1] -8.3796152  0.8963473
```

Die analoge Situation mit zwei abhängigen Stichproben lässt sich auf den Fall einer Stichprobe zurückführen, indem eine Differenzvariable aus der jeweils pro Person berechneten Differenz beider Beobachtungen gebildet wird (vgl. Abschn. 7.2.3).

11.1.4 Lineare Modelle: case resampling

Die in Abschn. 6.2.2 und 6.3 vorgestellten Tests der Parameter einer linearen Regression setzen die Gültigkeit verschiedener Annahmen voraus, deren Plausibilität mit Methoden untersucht werden kann, wie sie Abschn. 6.5 erläutert. Sind diese Annahmen verletzt, sind die berechneten Standardfehler der Parameterschätzer womöglich verzerrt und führen zu falschen p -Werten. Oft eignen sich in diesem Fall Bootstrap-Verfahren, um angemessene Vertrauensintervalle für die Regressionsgewichte zu erhalten.

Als Beispiel sei auf die Daten der multiplen linearen Regression in Abschn. 6.3.1 zurückgegriffen, die das Körpergewicht anhand der Prädiktoren Körpergröße, Alter und der für Sport aufgewendeten Zeit vorhersagen soll. Die Daten wurden unter gültigen Modellannahmen simuliert, was es hier erlaubt, die korrekten Standardfehler und Konfidenzintervalle der parametrischen Regressionsanalyse zur Validierung der Bootstrap-Ergebnisse zu verwenden.

```
> sqrt(diag(vcov(fitHAS)))          # parametrische Standardfehler
(Intercept)      height       age      sport
 8.18162620  0.04590320  0.04033012  0.01201414

> confint(fitHAS)                  # parametrische Konfidenzintervalle
           2.5 %    97.5 %
(Intercept) -8.1391666  24.3416327
height       0.4237764  0.6060106
age         -0.3667003 -0.2065910
sport        -0.4398003 -0.3921046
```

Die erste Methode, bootstrapping auf die Situation eines linearen Modells wie das der Regression anzuwenden, besteht im *case resampling*, auch *Vektor-Sampling* genannt: Hierfür werden

die beobachteten Werte aller Variablen als zufällig betrachtet, insbesondere auch jene der Prädiktoren. Für jede Replikation wird nun aus der Menge der beobachteten Personen (*cases*) mit Zurücklegen eine Stichprobe vom ursprünglichen Umfang gezogen. Die Werte der jeweils gezogenen Personen für Prädiktoren und Kriterium liegen der Anpassung des Regressionsmodells für eine Replikation zugrunde. So liefert jede Replikation eine Schätzung für jeden Regressionsparameter, woraus sich deren Bootstrap-Verteilungen – und damit Standardfehler und Konfidenzintervalle bestimmen lassen.

Die Methode gilt als robust, da sich mit jeder Replikation die Design-Matrix des Modells ändert und daher Ausreißer oder übermäßig einflussreiche Beobachtungen die Parameterschätzungen nicht immer verzerrn. Case resampling ist auch für verallgemeinerte lineare Modelle geeignet (vgl. Abschn. 8).

Die im Aufruf von `boot()` für das Argument `statistic` genannte Funktion muss unter `data` einen Datensatz mit den ursprünglichen Werten von Prädiktoren und Kriterium akzeptieren sowie die Parameterschätzungen für die über den Indexvektor `idx` definierte Replikation zurückgeben.

```
# berechne für Daten dat und Replikation idx die Regressionsgewichte
> getRegr <- function(dat, idx) {
+   bsFit <- lm(weight ~ height + age + sport, subset=idx, data=dat)
+   coef(bsFit)                                # Regressionsgewichte Replikation
+ }

> library(boot)                               # für boot(), boot.ci()
> nR <- 999                                    # Anzahl BS-Replikationen
> (bsRegr <- boot(regrDf, statistic=getRegr, R=nR))
ORDINARY NONPARAMETRIC BOOTSTRAP
Call:
boot(data = regrDf, statistic = getRegr, R = nR)

Bootstrap Statistics :
      original       bias    std. error
t1*   8.1012331 -0.1025429121  7.86996801
t2*   0.5148935  0.0001543189  0.04557696
t3*  -0.2866457  0.0017596402  0.04037238
t4*  -0.4159525  0.0004067643  0.01125020
```

Die Zeilen der Ausgabe nennen die Bootstrap-Kennwerte für jeweils einen Koeffizienten in der Reihenfolge des Rückgabewertes der im Aufruf von `boot()` für `statistic` verwendeten Funktion. Für die gewählten Daten stimmen die Bootstrap-Standardfehler weitgehend mit jenen der parametrischen Regressionsanalyse überein. Das Vertrauensintervall für jeden Parameter liefert wieder `boot.ci()`, wobei mit dem Argument `index=(Nummer)` auszuwählen ist, für welchen Parameter θ_j das Vertrauensintervall benötigt wird.

```
# BCa-Konfidenzintervalle für die Regressionsparameter
> boot.ci(bsRegr, conf=0.95, type="bca", index=1)$bca      # b0
[1,] 0.95 22.45 972.23 -6.975915 24.44877
```

```
> boot.ci(bsRegr, conf=0.95, type="bca", index=2)$bca      # height
[1,] 0.95 27.62 977.4 0.4238508 0.5991131

> boot.ci(bsRegr, conf=0.95, type="bca", index=3)$bca      # age
[1,] 0.95 26.18 976.23 -0.3637959 -0.2059037

> boot.ci(bsRegr, conf=0.95, type="bca", index=4)$bca      # sport
[1,] 0.95 22.45 972.22 -0.4382711 -0.3939868
```

Das erste Element des Vektors in der Komponente `bca` der von `boot.ci()` zurückgegebenen Liste nennt die Breite des Intervalls, die beiden folgenden Elemente die zu den $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantilen gehörenden Indizes für den Vektor der sortierten Werte,³ die letzten beiden Elemente sind schließlich die gesuchten Intervallgrenzen. Auch die Konfidenzintervalle gleichen hier jenen aus der parametrischen Regressionsanalyse.

11.1.5 Lineare Modelle: model-based resampling

Varianzanalysen lassen sich wie eine Regression als lineares Modell formulieren, wobei die Gruppenzugehörigkeiten (Werte der UV) die Rolle der Prädiktoren einnehmen und eine Zielvariable (AV) das Kriterium bildet (vgl. Abschn. 12.9). Die Gruppenzugehörigkeiten sind im Gegensatz zu den Prädiktorwerten der Regression experimentell festgelegt und sollten deshalb für alle resamples konstant sein. Eine Methode, um dies zu gewährleisten, besteht im *model-based resampling*. Hier werden nur die Werte der AV als mit zufälligen Fehlern behaftet betrachtet. Dieser Logik folgend berechnet man zunächst für die Daten der Basisstichprobe die Modellvorhersage \hat{Y} sowie die zugehörigen Residuen $E = Y - \hat{Y}$.

Die Residuen werden daraufhin zu $\frac{E}{\sqrt{1-h}}$ reskaliert, damit sie überall die bedingte theoretische Streuung σ besitzen (vgl. Abschn. 12.9.4). Dabei ist h die Variable Hebelwert (vgl. Abschn. 6.5.1, 6.5.2). Um zu gewährleisten, dass E im Mittel 0 ist, sollte das Modell einen absoluten Term β_0 einschließen oder zentrierte Variablen verwenden.

Für jede Replikation wird aus $\frac{E}{\sqrt{1-h}}$ mit Zurücklegen eine Stichprobe E^* vom ursprünglichen Umfang gezogen. Diese Resample-Residuen werden zu \hat{Y} addiert, um die Resample-AV $Y^* = \hat{Y} + E^*$ zu erhalten. Für jede Bootstrap-Schätzung der Parameter wird dann die Varianzanalyse mit Y^* und der ursprünglichen UV berechnet. Model-based resampling gilt als effizienter als case resampling, allerdings auch als anfälliger für eine falsche Modell-Spezifizierung.

Residuen E und Modellvorhersage \hat{Y} für die Basisstichprobe lassen sich sowohl für das H_0 - wie für das H_1 -Modell bilden: Unter der H_0 der einfaktoriellen Varianzanalyse unterscheiden sich die Erwartungswerte in den Gruppen nicht, als Vorhersage \hat{Y} ergibt sich damit für alle Personen der Gesamtmittelwert M . Wird dieses Modell mit der Formel $(AV) \sim 1$ für die Basisstichprobe angepasst, erzeugt das bootstrapping eine Approximation der Verteilung von $\hat{\theta}$ (etwa des

³Die Indizes sind hier trotz der 999 Replikationen nicht ganzzahlig (25 und 975), da die dem BC_a -Intervall zugrundeliegende Korrektur über die Verschiebung der Intervallgrenzen funktioniert. Vergleiche etwa das Perzentil-Intervall für θ_1 aus `boot.ci(bsRegr, conf=0.95, type="perc", index=1)$percent`.

F -Bruchs) unter H_0 . Dies erlaubt es, p -Werte direkt als Anteil der resamples zu berechnen, bei denen $\hat{\theta}^*$ i.S. der H_1 mindestens so extrem wie $\hat{\theta}$ ist.⁴

Als Beispiel sei auf die Daten der einfaktoriellen Varianzanalyse in Abschn. 7.3.1 zurückgegriffen. Die Daten wurden unter gültigen Modellannahmen simuliert, was es hier erlaubt, die Verteilung der Bootstrap-Schätzer F^* mit der F -Verteilung zu vergleichen und den Bootstrap- p -Wert mit dem p -Wert der F -Verteilung zu validieren.

```
> anBase <- anova(lm(DV ~ IV))          # ANOVA Basisstichprobe
> Fbase  <- anBase["IV", "F value"]       # F-Wert Basisstichprobe
> (pBase <- anBase["IV", "Pr(>F)"])      # p-Wert Basisstichprobe
[1] 0.002921932

# H0-Modell in Basisstichprobe anpassen
> fit0 <- lm(DV ~ 1)                      # Modellanpassung
> E     <- residuals(fit0)                 # ursprüngliche Residuen
> Er    <- E / sqrt(1-hatvalues(fit0))      # reskalierte Residuen
> Yhat <- fitted(fit0)                     # ursprüngliche Vorhersage

# ANOVA-Parameter für ursprüngliche Gruppenzugehörigkeiten und Y*
> getAnova <- function(dat, idx) {
+   Ystar <- Yhat + Er[idx]                  # Resample-AV Y* = Y^ + E*
+   anBS  <- anova(lm(Ystar ~ IV, data=dat))  # ANOVA Resample-AV
+   anBS["IV", "F value"]                    # F*-Wert Replikation
+ }

> library(boot)                            # für boot(), boot.ci()
> nR      <- 999                          # Anzahl Replikationen
> (bsAnova <- boot(dfCRp, statistic=getAnova, R=nR))
ORDINARY NONPARAMETRIC BOOTSTRAP
Call:
boot(data = dfCRp, statistic = getAnova, R = nR)

Bootstrap Statistics :
        original     bias   std. error
t1*  4.861867 -3.887228   0.8136521
```

Der für den p -Wert notwendige Vergleich $F^* \geq F$ wird hier in zwei Vergleiche aufgeteilt, um robuster gegenüber Problemen der numerischen Repräsentation von Gleitkommazahlen zu sein (vgl. Abschn. 1.3.6). Dafür wird nur auf ungefähre, nicht auf exakte Gleichheit von F^* und F geprüft.

```
# F* aus Bootstrap-Replikationen
> Fstar <- bsAnova$t
```

⁴Der p -Wert kann bei Monte-Carlo-Approximationen zur höheren Genauigkeit nach Hinzufügen eines zusätzlichen extremeren Falles gebildet werden: Ist n_R die Anzahl der generierten resamples und n^* die Anzahl der Fälle, bei denen $\hat{\theta}^*$ mindestens so extrem wie $\hat{\theta}$ ist, setzt man $p = \frac{n^*+1}{n_R+1}$. Auf diese Weise wird vermieden, dass der p -Wert exakt 0 werden kann.

```
# numerische Toleranz wg. Gleitkomma-Arithmetik
> tol <- .Machine$double.eps^0.5

# Fstar größer Basis-F ODER Fstar ungefähr gleich Basis-F?
> FsIsGEQ <- (Fstar > Fbase) | (abs(Fstar - Fbase) < tol)

# p-Wert: Anteil der mindestens so extremen F* wie Basis F
> (pValBS <- (sum(FsIsGEQ) + 1) / (length(Fstar) + 1))
[1] 0.004

# Grafik: kumulierte relative Häufigkeiten von F*
> plot(Fstar, ecdf(Fstar)(Fstar), col="gray60", pch=1,
+       xlab="f* bzw. f", ylab="P(F <= f)",
+       main="F*: Kumulierte rel. Häufigkeiten und Verteilungsfunktion")

# theoretische Verteilungsfunktion von t und Legende
> curve(pf(x, P-1, sum(Nj) - P), lwd=2, add=TRUE)
> legend(x="topleft", lty=c(NA, 1), pch=c(1, NA), lwd=c(2, 2),
+         col=c("gray60", "black"), legend=c("F*", "F"))
```

Die Verteilung der F^* ist hier der theoretischen F -Verteilung sehr ähnlich (Abb. 11.2), was auch für die Größenordnung des Bootstrap- p -Wertes verglichen mit dem p -Wert der ursprünglichen Varianzanalyse gilt.

11.1.6 Lineare Modelle: wild bootstrap

Eine Variante des model-based resampling ist der *wild bootstrap* für Situationen, in denen Heteroskedastizität vorliegt. Hier wird E^* aus dem Produkt $\frac{E}{\sqrt{1-h}} \cdot U$ gebildet. Dabei ist U eine unabhängige Zufallsvariable mit $E(U) = 0$ und $E(U^2) = 1$, deren Werte für jede Replikation simuliert werden müssen.

- Eine Wahl für U sind dichotome Variablen mit F_1 -Verteilung, die den Wert $\frac{-(\sqrt{5}-1)}{2}$ mit Wahrscheinlichkeit $p = \frac{\sqrt{5}+1}{2\sqrt{5}}$ annehmen und den Wert $\frac{\sqrt{5}+1}{2}$ mit Wahrscheinlichkeit $1-p = \frac{\sqrt{5}-1}{2\sqrt{5}}$.
- Eine alternative Wahl für U sind ebenfalls dichotome Variablen mit F_2 - bzw. Rademacher-Verteilung: Sie nehmen die Werte -1 und 1 jeweils mit Wahrscheinlichkeit $\frac{1}{2}$ an, drehen also das ursprüngliche Vorzeichen jedes Residuums zufällig um.

```
> getAnovaWild <- function(dat, idx) {
+   n <- length(idx)                      # Größe der Replikation
+   # Ur: Rademacher Variablen, Uf: zweite Variante
+   Ur <- sample(c(-1, 1), size=n, replace=TRUE, prob=c(0.5, 0.5))
+   Uf <- sample(c(-(sqrt(5)-1)/2, (sqrt(5)+1)/2), size=n,
+               replace=TRUE, prob=c((sqrt(5)+1)/(2*sqrt(5)),
+                                     (sqrt(5)-1)/(2*sqrt(5))))
```

```

+
+   Ystar <- Yhat + (Er*Ur)[idx]          # E* mit Rademacher-Variable
+   # Ystar <- Yhat + (Er*Uf)[idx]          # zweite Variante
+   anBS  <- anova(lm(Ystar ~ IV, data=dat))    # ANOVA Resample-AV
+   anBS["IV", "F value"]                  # F*-Wert Replikation
}

# fortfahren wie oben
> bsAnovaW <- boot(dfCRp, statistic=getAnovaWild, R=nR)
> FstarW   <- bsAnova$statistic           # F*-Werte
> tol      <- .Machine$double.eps^0.5     # numerische Toleranz

# Fstar größer Basis-F ODER Fstar ungefähr gleich Basis-F?
> FsIsGEQ  <- (FstarW > Fbase) | (abs(FstarW - Fbase) < tol)
> (pValBSw <- (sum(FsIsGEQ) + 1) / (length(FstarW) + 1))    # ...

```

Werden Vorhersage und Residuen für die Basisstichprobe für das H_1 Modell mit der Formel $\langle AV \rangle \sim \langle UV \rangle$ berechnet, erzeugt das modellbasierte bootstrapping eine Verteilung von $\hat{\theta}^*$, aus der sich Vertrauensintervalle für θ bestimmen lassen (vgl. Abschn. 11.1.2, 11.1.4). Für die Bootstrap-Verteilung von F^* sollte dabei auf das BC_a -Intervall zurückgegriffen werden, da diese Verteilung typischerweise schief ist und der Schätzer einen deutlichen bias aufweist.

11.2 Permutationstests

Permutationstests (Chihara & Hesterberg, 2011; Good, 2004) verwenden analog zu Bootstrap-Verfahren Permutationen einer festen Basisstichprobe als resamples, um ausgehend von der empirischen Verteilung der für diese resamples berechneten Schätzer $\hat{\theta}^*$ Aussagen über die theoretische Verteilung einer Teststatistik $\hat{\theta}$ abzuleiten. Jede Permutation wird dabei so gebildet, dass sie dieselben Werte der Basisstichprobe umfasst, die Reihenfolge der Beobachtungen i. S. der Zusammensetzung der Untersuchungsbedingungen jedoch im Einklang mit der H_0 des Tests im gegebenen Untersuchungsdesign variiert.

Stimmen etwa unter H_0 die Verteilungen einer Zielvariable (AV) in zwei Bedingungen überein, ist die Zugehörigkeit der Beobachtungen zu einer Bedingung für die Ausprägung der AV unwesentlich und kann deshalb permutiert werden – jede Beobachtung hätte genauso gut aus jeder Bedingung stammen können.⁵ Im Fall von abhängigen Stichproben ist dies separat innerhalb jedes Beobachtungsobjekts zu tun, bei unabhängigen Stichproben entsprechend über Beobachtungsobjekte hinweg. Sind zwei Variablen unter H_0 unabhängig, ist es für die Ausprägung der zweiten AV irrelevant, welchen Wert die erste AV besitzt – die Zuordnung von Werten der zweiten AV zu jenen der ersten kann also permutiert werden.

Die empirische Verteilung von $\hat{\theta}^*$ schätzt die Verteilung von $\hat{\theta}$ unter H_0 , was es erlaubt, p -Werte direkt zu berechnen: Dies ist der Anteil der Permutationen, bei denen $\hat{\theta}^*$ i. S. der H_1 mindestens so extrem wie $\hat{\theta}$ ist. Da die Anzahl möglicher Permutationen (und damit der Rechenaufwand) sehr schnell mit der Stichprobengröße wächst, ist es nur in Spezialfällen oder bei kleinen

⁵Formal muss das Kriterium der *Austauschbarkeit* erfüllt sein (Good, 2004).

Stichproben möglich, die empirische Verteilung von $\hat{\theta}^*$ exakt zu bestimmen. Ist die praktische Berechenbarkeit aller $\hat{\theta}^*$ nicht gegeben, lassen sich Permutationstests als Monte-Carlo-Verfahren durchführen, die $\hat{\theta}^*$ nur für eine zufällige Auswahl von Permutationen berechnen (vgl. Abschn. 11.1.5, Fußnote 4).⁶

Das bereits in Abschn. 10.5.8, 10.5.10 und 10.5.11 verwendete Paket `coin` stellt für viele Hypothesen exakte, oder aber durch Monte-Carlo-Verfahren approximierte Permutationstests bereit, für deren konventionelle Prüfung sonst auf parametrische Tests oder nonparametrische Verfahren mit einer asymptotisch gültigen Verteilung der Teststatistik zurückgegriffen werden muss – für eine Übersicht vgl. `vignette("coin")`.⁶

11.2.1 Test auf gleiche Lageparameter in unabhängigen Stichproben

Im Beispiel soll linksseitig getestet werden, ob die Verteilung einer quantitativen AV in (hier zwei) unabhängigen Stichproben übereinstimmt. Bei Verteilungen gleicher Form ist dies der Fall, wenn ihre Lageparameter identisch sind. Anders als etwa beim Wilcoxon-Rangsummen-Test (vgl. Abschn. 10.5.4) sollen hier die ursprünglichen AV-Werte ohne Rangtransformation Verwendung finden, wofür sich `oneway_test()` aus dem Paket `coin` eignet.

```
> oneway_test(formula=<Modellformel>, data=<Datensatz>, subset=<Indexvektor>,
+             alternative=c("two.sided", "less", "greater"),
+             distribution=<Verteilungstyp>)
```

Unter `formula` sind Daten und Gruppierungsvariable als Modellformel $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$ einzugeben, wobei $\langle \text{UV} \rangle$ ein Faktor derselben Länge wie $\langle \text{AV} \rangle$ ist und für jede Beobachtung die zugehörige UV-Stufe angibt. Geschieht dies mit Variablen aus einem Datensatz, muss dieser unter `data` eingetragen werden. Das Argument `subset` erlaubt es, nur eine Teilmenge der Fälle einfließen zu lassen, es erwartet einen entsprechenden Indexvektor, der sich auf die Zeilen des Datensatzes bezieht. Mit `alternative` wird festgelegt, ob die H_1 gerichtet ("`less`" bzw. "`greater`") oder ungerichtet ("`two.sided`") ist. Die Anzahl der zufälligen Permutationen, auf deren Basis die Verteilung der Teststatistik approximiert wird, lässt sich über `approximate(B=<Anzahl>)` für das Argument `distribution` übergeben. Mit der in bestimmten Situationen wählbaren Option "`exact`" für dieses Argument basiert der p -Wert auf der exakten Verteilung aller möglichen $\hat{\theta}^*$.

```
> Nj      <- c(7, 8)                                # Gruppengrößen
> DVa    <- round(rnorm(Nj[1], 100, 20))          # Daten Gruppe A
> DVb    <- round(rnorm(Nj[2], 110, 20))          # Daten Gruppe B
> DVab   <- c(DVa, DVb)                            # Gesamt-Daten
> IVbtw <- factor(rep(c("A", "B"), Nj))           # Gruppenzugehörigkeit
> library(coin)                                     # für oneway_test()
> (ot <- oneway_test(DVab ~ IVbtw, alternative="less",
+                     distribution="exact"))
Exact 2-Sample Permutation Test
data: DVab by IVbtw (A, B)
```

⁶Auch die Pakete `resample` und `vegan` (Oksanen et al., 2014) bieten flexible Möglichkeiten, um Permutationstests für verschiedenen Untersuchungs-Designs umzusetzen.

```
Z = -2.0981, p-value = 0.01756
alternative hypothesis: true mu is less than 0
```

Das Ergebnis nennt den Wert der intern verwendeten Teststatistik unter Z,⁷ gefolgt vom *p*-Wert unter *p-value*.

Die Hilfe-Seite von **support()** erläutert die Verwendung von **dperm()** und **qperm()**, um Dichteverteilung und Quantile der Permutations-Teststatistik von **oneway_test()** zu ermitteln. Die Permutationsverteilung sollte unimodal und symmetrisch sein, nicht unähnlich einer Normalverteilung (Abb. 11.3).

```
# Dichteverteilung der Teststatistik von oneway_test()
> supp <- support(ot)
> dens <- sapply(supp, dperm, object=ot)           # Dichte
> plot(supp, dens, xlab="Support", ylab=NA, pch=20,
+       main="Dichte Permutationsverteilung")

# Q-Q-Plot der Teststatistik von oneway_test() gegen Standard-NV
> qEmp <- sapply(ppoints(supp), qperm, object=ot)   # Quantile Teststat.
> qqnorm(qEmp, xlab="Quantile Normalverteilung",
+         ylab="Permutations-Quantile",
+         main="Permutations- vs. theoretische NV-Quantile")

> abline(a=0, b=1, lwd=2, col="blue")
```

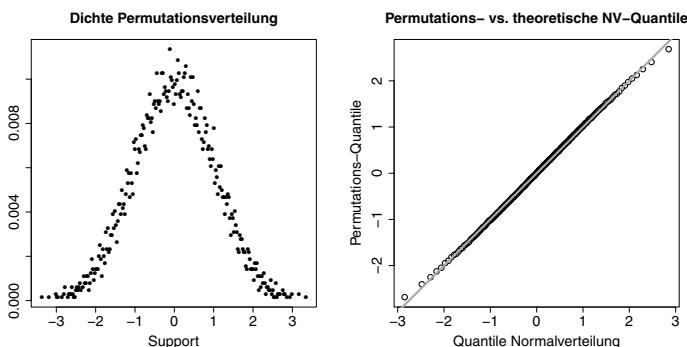


Abbildung 11.3: Dichteverteilung der Teststatistik des Permutationstests. Q-Q-Plot der Teststatistik des Permutationstests im Vergleich zur Standardnormalverteilung.

Zum Vergleich mit dem Permutationstest soll zunächst der *p*-Wert des analogen parametrischen *t*-Tests ermittelt werden (vgl. Abschn. 7.2.2). Bei der folgenden manuellen Kontrolle dient die Mittelwertsdifferenz zwischen beiden Gruppen als Teststatistik. Dabei ist zu beachten, dass $\hat{\theta}^*$ hier nicht für alle $N!$ möglichen Permutationen der Gesamtstichprobe vom Umfang N bestimmt werden muss. Dies ist nur für jene Permutationen notwendig, die auch zu unterschiedlichen

⁷Für deren Wahl vgl. `vignette("coin_implementation")`.

Gruppenzusammensetzungen führen, also nicht lediglich innerhalb jeder Gruppe die Reihenfolge vertauschen. Es gibt $\binom{N}{n_1}$ viele Möglichkeiten (Kombinationen, vgl. Abschn. 2.3.3), zwei Gruppen der Größe n_1 und n_2 zu bilden, dabei ist jede Kombination gleich wahrscheinlich.

```
# Vergleich: p-Wert aus analogem parametrischen t-Test
> t.test(DVab ~ IVbtw, alternative="less", var.equal=TRUE)$p.value
[1] 0.01483469

# Indizes aller unterschiedlichen Zusammensetzungen Gruppe A*
> idx <- seq(along=DVab) # Indizes Gesamt-Daten
> idxA <- combn(idx, Nj[1]) # alle n1-Kombinationen

# Mittelwertsdifferenz für gegebene Indizes x der Gruppe A*
> getDM <- function(x) { mean(DVab[x]) - mean(DVab[!(idx %in% x)]) }
> DMstar <- apply(idxA, 2, getDM) # M-Differenz aller Kombinationen
> DMbase <- mean(DVa) - mean(DVb) # beobachtete M-Differenz
```

Der für den p -Wert notwendige Vergleich $\hat{\theta}^* \leq \hat{\theta}$ wird hier in zwei Vergleiche aufgeteilt, um robuster gegenüber Problemen der numerischen Repräsentation von Gleitkommazahlen zu sein (vgl. Abschn. 1.3.6). Dafür wird nur auf ungefähre, nicht auf exakte Gleichheit von $\hat{\theta}^*$ und $\hat{\theta}$ geprüft.

```
> tol <- .Machine$double.eps^0.5 # numerische Toleranz

# DMstar kleiner Basis-DM ODER DMstar ungefähr gleich Basis-DM?
> DMsIsLEQ <- (DMstar < DMbase) | (abs(DMstar - DMbase) < tol)

# p-Wert: Anteil der mind. so extremen Mittelwertsdifferenzen
> (pVal <- sum(DMsIsLEQ) / length(DMstar))
[1] 0.01756022
```

11.2.2 Test auf gleiche Lageparameter in abhängigen Stichproben

Beim Test auf Übereinstimmung von Verteilungen aus (hier zwei) abhängigen Stichproben hat die für `oneway_test()` anzugebende Modellformel die Form $\langle AV \rangle \sim \langle UV \rangle + \langle BlockId \rangle$. Dabei codiert $\langle UV \rangle$ als Faktor derselben Länge wie $\langle AV \rangle$ den Messzeitpunkt. $\langle BlockId \rangle$ ist ebenfalls ein solcher Faktor und gibt im Fall von Messwiederholung die Zugehörigkeit jedes Wertes zu einem Beobachtungsobjekt (bei gematchten Personen: zu einem Block) an.

```
> N      <- 12 # Anzahl Personen
> id     <- factor(rep(1:N, times=2)) # Faktor Personen-ID
> DVpre  <- round(rnorm(N, 100, 20)) # Daten Zeitpunkt prä
> DVpost <- round(rnorm(N, 110, 20)) # Daten Zeitpunkt post
> DVpp   <- c(DVpre, DVpost) # Gesamt-Daten

# Faktor Messzeitpunkt
> IV <- factor(rep(0:1, each=N), labels=c("pre", "post"))
```

```
> library(coin)                                # für oneway_test()
> oneway_test(DVpp ~ IV | id, alternative="less",
+              distribution=approximate(B=9999))
Approximative 2-Sample Permutation Test
data: DVpp by IV (pre, post) stratified by id
Z = -0.6056, p-value = 0.2748
alternative hypothesis: true mu is less than 0
```

Das Ergebnis soll zunächst mit dem p -Wert des analogen parametrischen t -Tests verglichen werden (vgl. Abschn. 7.2.3). Bei der anschließenden manuellen Kontrolle dient die mittlere paarweise Differenz zwischen den Werten der zwei abhängigen Bedingungen als Teststatistik. Dafür ist separat für jede Person die Zuordnung ihrer beiden Werte zu einem Testzeitpunkt zu permutieren. Dies ist äquivalent zur Permutation des Vorzeichens der personenweisen Messwertdifferenz. Bei n Personen führt dies zu 2^n verschiedenen Gesamt-Permutationen.

```
> t.test(DVpp ~ IV, alternative="less", paired=TRUE)$p.value
[1] 0.2839126
```

```
# alle 2^N Möglichk., pro Person Vorzeichen der Differenz zu permutieren
> DVd    <- DVpre - DVpost      # personenweise Messwertdifferenzen
> ordLst <- lapply(numeric(N), function(x) { c(-1, 1) } )
> ordMat <- data.matrix(expand.grid(ordLst))    # alle 2^N Perm.

# für Gesamt-Permutation x der Vorzeichen: mittlere personenweise Diff.
> getMD  <- function(x) { mean(abs(DVd) * x) }
> MDstar <- apply(ordMat, 1, getMD)      # mittlere Differenzen alle Perm.
> MDbase <- mean(DVd)                  # mittl. Differenz Basis-Stichprobe
```

Der für den p -Wert notwendige Vergleich $\hat{\theta}^* \leq \hat{\theta}$ wird hier in zwei Vergleiche aufgeteilt, um robuster gegenüber Problemen der numerischen Repräsentation von Gleitkommazahlen zu sein (vgl. Abschn. 1.3.6). Dafür wird nur auf ungefähre, nicht auf exakte Gleichheit von $\hat{\theta}^*$ und $\hat{\theta}$ geprüft.

```
> tol <- .Machine$double.eps^0.5          # numerische Toleranz

# MDstar kleiner Basis-MD ODER MDstar ungefähr gleich Basis-MD?
> MDsIsLEQ <- (MDstar < MDbase) | (abs(MDstar - MDbase) < tol)

# p-Wert: Anteil der mind. so extremen personenweisen Differenzen
> (pVal <- sum(MDsIsLEQ) / length(MDstar))
[1] 0.2800293
```

11.2.3 Test auf Unabhängigkeit von zwei Variablen

Für den Test auf Unabhängigkeit zweier an denselben Personen erhobener Variablen sollen hier dichotome Daten dienen, um das Ergebnis der manuellen Umsetzung mit jenem von Fishers exaktem Test vergleichen zu können – dem passenden Permutationstest (vgl. Abschn. 10.2.4).

```
> Nf <- 8                                     # Anzahl Personen
> DV1 <- rbinom(Nf, size=1, prob=0.5)        # Daten AV 1
> DV2 <- rbinom(Nf, size=1, prob=0.5)        # Daten AV 2

# p-Wert rechtsseitiger Test
> fisher.test(DV1, DV2, alternative="greater")$p.value
[1] 0.7857143
```

Die manuelle Kontrolle verwendet `Permnn(<Vektor>)` aus dem Paket `DescTools`, um die Zuordnung von Werten der zweiten AV zu jenen der ersten zu permutieren. Die Funktion erzeugt eine Matrix mit allen $n!$ vielen Permutationen des übergebenen Vektors in den Zeilen. Mit den Indizes für den Vektor der zweiten AV liefern diese Permutationen die gewünschten Zuordnungen und führen so zu allen möglichen Kontingenztafeln der gemeinsamen Häufigkeiten mit denselben Randhäufigkeiten wie in der Basisstichprobe. Teststatistik ist die Anzahl der in der Diagonale einer Kontingenztafel stehenden Fälle, also der Übereinstimmungen beider Variablen.

```
# generiere alle Nf! vielen Zuordnungen von AV 1 zu AV 2
> library(DescTools)                         # für Permnn()

# Matrix aller Permutationen der Indizes 1:Nf
> permIdx <- Permnn(seq(length.out=Nf))

# Anzahl der Übereinstimmungen für gegebene Permutation x der AV 2
> getAgree <- function(x) { sum(diag(table(DV1, DV2[x]))) }

# Anzahl der Übereinstimmungen für jede der Nf! Permutationen
> resAgree <- apply(permIdx, 1, getAgree)
> agree12  <- sum(diag(table(DV1, DV2)))      # beobachtete Übereinst.

# p-Wert: Anteil der Perm. mit mind. so großer Übereinstimmung
> (pVal <- sum(resAgree >= agree12) / length(resAgree))
[1] 0.7857143
```

Kapitel 12

Multivariate Verfahren

Liegen von Beobachtungsobjekten Werte mehrerer Variablen vor, kann sich die Datenanalyse nicht nur auf jede Variable einzeln, sondern auch auf die gemeinsame Verteilung der Variablen beziehen. Solche Fragestellungen sind mit multivariaten Verfahren zu bearbeiten (Backhaus, Erichson, Plinke & Weiber, 2011; Härdle & Simar, 2012; Mardia, Kent & Bibby, 1980), deren Anwendung in R Everitt und Hothorn (2011) vertiefend behandeln. Abschnitt 14.6.8 und 14.8 thematisieren Möglichkeiten, multivariate Daten in Diagrammen zu veranschaulichen.

12.1 Lineare Algebra

Vielen statistischen Auswertungen – insbesondere im Bereich multivariater Verfahren – liegt im Kern das Rechnen mit Matrizen zugrunde, weshalb an dieser Stelle auf einige grundlegende Funktionen zur Matrix-Algebra eingegangen werden soll. Diese sind jedoch nur für spätere manuelle Kontrollrechnungen relevant, nicht aber für die reine Anwendung multivariater Tests mit R-eigenen Funktionen.¹ Die Rechentechniken und zugehörigen Konzepte der linearen Algebra (Fischer, 2010; Strang, 2003) seien als bekannt vorausgesetzt.

12.1.1 Matrix-Algebra

Eine Matrix \mathbf{X} wird mit `t(Matrix)` transponiert. Hierdurch werden die Zeilen von \mathbf{X} zu den Spalten der Transponierten \mathbf{X}^t sowie entsprechend die Spalten von \mathbf{X} zu Zeilen von \mathbf{X}^t , und es gilt $(\mathbf{X}^t)^t = \mathbf{X}$.

```
> N <- 4                                # Anzahl Zeilen
> p <- 2                                # Anzahl Spalten
> (X <- matrix(c(20, 26, 10, 19, 29, 27, 20, 12), nrow=N, ncol=p))
 [,1] [,2]
[1,] 20   29
[2,] 26   27
[3,] 10   20
[4,] 19   12
```

¹Die hier vorgestellten Rechenwege setzen die mathematischen Formeln direkt um. Tatsächlich gibt es häufig numerisch effizientere und stabilere Möglichkeiten, um dieselben Ergebnisse zu erhalten. So entspricht die Implementierung von R-eigenen Funktionen auch meist nicht den hier vorgestellten Rechnungen (Bates, 2004). Vergleiche Abschn. 15.3.3 für Wege, die Effizienz der Berechnungen zu steigern. Das Paket `Matrix` (Bates & Maechler, 2014) enthält fortgeschrittene Methoden der Matrix-Algebra – etwa zu schwachbesetzten Matrizen, die u. a. als Designmatrizen linearer Modelle auftauchen (vgl. Abschn. 12.9.1).

```
> t(X)                                # Transponierte
      [,1]  [,2]  [,3]  [,4]
[1,]    20    26    10    19
[2,]    29    27    20    12
```

Die Diagonalelemente einer Matrix gibt `diag(<Matrix>)` in Form eines Vektors aus.² Die Umkehrung `diag(<Vektor>)` erzeugt eine Diagonalmatrix, deren Diagonale aus den Elementen von `<Vektor>` besteht. Besitzt der übergebene Vektor allerdings nur ein Element, kommt die Variante `diag(<Anzahl>)` zum tragen, die als besondere Diagonalmatrix eine Einheitsmatrix mit `<Anzahl>` vielen Zeilen und Spalten liefert.³

```
# Diagonale der Kovarianzmatrix von X -> Varianzen der Spalten von X
```

```
> diag(cov(X))
```

```
[1] 43.58333 59.33333
```

```
> diag(1:3)                                # Diagonalmatrix mit Diagonale 1, 2, 3
```

```
      [,1]  [,2]  [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

```
> diag(2)                                  # 2x2 Einheitsmatrix
```

```
      [,1]  [,2]
[1,]    1    0
[2,]    0    1
```

Da auch bei Matrizen die herkömmlichen Operatoren elementweise arbeiten, eignet sich für die Addition von Matrizen \mathbf{X} und \mathbf{Y} passender Dimensionierung der `+` Operator mit $\mathbf{X} + \mathbf{Y}$ genauso wie `*` für die Multiplikation von Matrizen mit einem Skalar mittels `<Matrix> * <Zahl>`. Auch $\mathbf{X} * \mathbf{Y}$ ist als elementweises Hadamard-Produkt $\mathbf{X} \circ \mathbf{Y}$ von gleich dimensionierten Matrizen zu verstehen. Die Matrix-Multiplikation einer $(p \times q)$ -Matrix \mathbf{X} mit einer $(q \times r)$ -Matrix \mathbf{Y} lässt sich dagegen mit $\mathbf{X} \%*\% \mathbf{Y}$ formulieren.⁴ Das Ergebnis ist die $(p \times r)$ -Matrix der Skalarprodukte der Zeilen von \mathbf{X} mit den Spalten von \mathbf{Y} . Als Rechenregel kommt in vielen Situationen $(\mathbf{XY})^t = \mathbf{Y}^t \mathbf{X}^t$ zum tragen.

Für den häufig genutzten Spezialfall $\mathbf{X}^t \mathbf{Y}$ existiert die Funktion `crossprod(X, Y)`, die alle paarweisen Skalarprodukte der Spalten von \mathbf{X} und \mathbf{Y} berechnet.⁵ `crossprod(X)` ist die Kurzform für $\mathbf{X}^t \mathbf{X}$. Das Kronecker-Produkt $\mathbf{X} \otimes \mathbf{Y}$ erhält man mit der Funktion `kronecker()`, deren Operator-Schreibweise $\mathbf{X} \%x\% \mathbf{Y}$ lautet.

²Bei nicht quadratischen $(p \times q)$ -Matrizen \mathbf{X} sind dies die Elemente x_{11}, \dots, x_{pp} (für $p < q$) bzw. x_{11}, \dots, x_{qq} (für $p > q$).

³Eine Dezimalzahl wird dabei tranchiert. Eine (1×1) -Diagonalmatrix \mathbf{X} kann mit einem weiteren Argument als `diag(x, nrow=1)` erzeugt werden.

⁴Mit dem Operator `<Matrix> \%~% <Zahl>` aus dem Paket `expm` (Goulet et al., 2014) ist das Exponenzieren von quadratischen Matrizen möglich, mit `logm()` aus demselben Paket das Logarithmieren.

⁵Sie ist zudem numerisch effizienter als die Verwendung von `t(X) \%*\% Y`. Die Benennung erscheint unglücklich, da Verwechslungen mit dem Vektor-Kreuzprodukt naheliegen, das man stattdessen mit `cross()` aus dem Paket `pracma` (Borchers, 2014) erhält.

Die Matrix-Multiplikation ist auch auf Vektoren anwendbar, da diese als Matrizen mit nur einer Zeile bzw. einer Spalte aufgefasst werden können. Mit `c()` oder `numeric()` erstellte Vektoren werden in den meisten Rechnungen automatisch so in Zeilen- oder Spaltenvektoren konvertiert, dass die Rechenregeln erfüllt sind.

```
# c(1, 2) wie Zeilenvektor behandelt: (1x2) Vektor %*% (2x3) Matrix
> c(1, 2) %*% rbind(c(1, 2, 3), c(4, 5, 6))
[1] [2] [3]
[1,] 9   12  15

# c(7, 8, 9) wie Spaltenvektor behandelt: (2x3) Matrix %*% (3x1) Vektor
> rbind(c(1, 2, 3), c(4, 5, 6)) %*% c(7, 8, 9)
[1]
[1,] 50
[2,] 122
```

Das Skalarprodukt zweier Vektoren \mathbf{x} und \mathbf{y} ist also mittels `crossprod(x, y)`, oder aber mit `sum(x*y)` berechenbar. Das Ergebnis von `crossprod()` ist dabei immer eine Matrix, auch wenn diese nur aus einer Spalte oder einzelnen Zahl besteht.

Als Beispiel sollen einige wichtige Matrizen zu einer gegebenen $(n \times p)$ -Datenmatrix \mathbf{X} mit Variablen in den Spalten berechnet werden: Die spaltenweise zentrierte Datenmatrix $\dot{\mathbf{X}}$ führt zur SSP-Matrix $\dot{\mathbf{X}}^t \dot{\mathbf{X}}$ (auch SSCP-Matrix genannt, *sum of squares and cross products*).⁶ Sie ist das $(n - 1)$ -fache der korrigierten Kovarianzmatrix \mathbf{S} .

Mit Hilfe der aus den Kehrwerten der korrigierten Streuungen gebildeten Diagonalmatrix $\mathbf{D}_S^{-1/2}$ erhält man die zu einer Kovarianzmatrix \mathbf{S} gehörende Korrelationsmatrix als $\mathbf{R} = \mathbf{D}_S^{-1/2} \mathbf{S} \mathbf{D}_S^{-1/2}$. Hierfür eignet sich `cov2cor(<K>)`.

```
> (Xc <- diag(N) - matrix(rep(1/N, N^2), nrow=N))      # Zentriermatrix
[1] [2] [3] [4]
[1,] 0.75 -0.25 -0.25 -0.25
[2,] -0.25  0.75 -0.25 -0.25
[3,] -0.25 -0.25  0.75 -0.25
[4,] -0.25 -0.25 -0.25  0.75

> all.equal(Xc %*% Xc, Xc)                                # Zentriermatrix ist idempotent
[1] TRUE

> (Xdot <- Xc %*% X)                                     # spaltenweise zentrierte Daten
[1] [2]
[1,] 1.25    7
[2,] 7.25    5
[3,] -8.75   -2
[4,] 0.25   -10
```

⁶Im Anwendungsfall würde man stattdessen auf `scale(<Matrix>, center=TRUE, scale=FALSE)` zurückgreifen, um eine Matrix spaltenweise zu zentrieren.

```

> (SSP <- t(Xdot) %*% Xdot)           # SSP-Matrix
[1,] 130.75   60
[2,]  60.00  178

> crossprod(Xdot)
> (1/(N-1)) * SSP
[1,] 43.58333 20.00000
[2,] 20.00000 59.33333

> (S <- cov(X))                      # korrigierte Kovarianzmatrix
[1,] 43.58333 20.00000
[2,] 20.00000 59.33333

> Dsi <- diag(1/sqrt(diag(S)))       # Diagonalmatrix: 1 / Streuungen
> Dsi %*% S %*% Dsi                # Korrelationsmatrix
[1,] 1.0000000 0.3932968
[2,] 0.3932968 1.0000000

> cov2cor(S)                        # Kontrolle ...

```

Für Anwendungen, in denen die Spalten einer Matrix \mathbf{X} mit einem Vektor \mathbf{a} verrechnet werden müssen, eignet sich `sweep()`. So könnte \mathbf{a} etwa zu jeder Zeile von \mathbf{X} addiert oder \mathbf{X} spaltenweise normiert werden. Alternativ könnte letzteres auch mit $\mathbf{X}\mathbf{D}$ geschehen, wenn \mathbf{D} die Diagonalmatrix aus den Kehrwerten der Spaltenlängen ist.

```

> b <- 2                                # Skalar
> a <- c(-2, 1)                          # Vektor
> sweep(b * X, 2, a, "+")
[1,] 38 59
[2,] 50 55
[3,] 18 41
[4,] 36 25

> colLens <- sqrt(colSums(X^2))          # euklidische Längen der Spalten
> sweep(X, 2, colLens, "/")              # spaltenweise normierte Daten
[1,] 0.5101443 0.6307329
[2,] 0.6631876 0.5872341
[3,] 0.2550722 0.4349882
[4,] 0.4846371 0.2609929

# äquivalent: Mult. mit Diag.-Matrix aus Kehrwerten der Spaltenlängen

```

```
> X %*% diag(1/colLens) # ...
```

12.1.2 Lineare Gleichungssysteme lösen

Für die Berechnung der Inversen \mathbf{A}^{-1} einer quadratischen Matrix \mathbf{A} mit vollem Rang stellt R keine separate Funktion bereit. Aus diesem Grund ist die allgemeinere Funktion `solve(a=Matrix, b=Matrix)` zu nutzen, die die Lösung \mathbf{x} des linearen Gleichungssystems $\mathbf{Ax} = \mathbf{b}$ liefert, wobei für das Argument `a` eine invertierbare quadratische Matrix und für `b` ein Vektor oder eine Matrix passender Dimensionierung anzugeben ist. Fehlt das Argument `b`, wird als rechte Seite des Gleichungssystems die passende Einheitsmatrix \mathbf{I} angenommen, womit sich als Lösung für \mathbf{x} in $\mathbf{Ax} = \mathbf{I}$ die Inverse \mathbf{A}^{-1} ergibt.⁷

Für \mathbf{A}^{-1} gilt etwa $(\mathbf{A}^{-1})^{-1} = \mathbf{A}$, $(\mathbf{A}^{-1})^t = (\mathbf{A}^t)^{-1}$ und $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$, wenn \mathbf{B} ebenfalls eine invertierbare Matrix passender Dimensionierung ist.

```
> Y      <- matrix(c(1, 1, 1, -1), nrow=2)           # Matrix Y
> (Yinv <- solve(Y))                                # Inverse von Y
[1,] [2]
[1,] 0.5   0.5
[2,] 0.5   -0.5

> Y %*% Yinv                                         # Kontrolle: Einheitsmatrix
[1,] [2]
[1,] 1     0
[2,] 0     1

# löse lineares Gleichungssystem
> A <- matrix(c(9, 1, -5, 0), nrow=2)    # Matrix A
> b <- c(5, -3)                               # Ergebnisvektor
> (x <- solve(A, b))                         # Lösung x von Ax = b
[1] -3.0 -6.4

> A %*% x                                     # Kontrolle: reproduziere b
[1,]
[1,] 5
[2,] -3
```

12.1.3 Norm und Abstand von Vektoren und Matrizen

Das mit `crossprod(Vektor)` ermittelte Skalarprodukt $\mathbf{x}^t \mathbf{x}$ eines Vektors \mathbf{x} mit sich selbst liefert ebenso wie `sum(Vektor)^2` dessen quadrierte euklidische Länge $\|\mathbf{x}\|^2$ i. S. der Norm. Genauso eignen sich `colSums(Matrix)^2` und `crossprod(Matrix)`, um die quadrierten Spaltennormen einer Matrix zu ermitteln, die dort im Ergebnis in der Diagonale stehen.

⁷Für die Pseudoinverse \mathbf{A}^+ einer nicht invertierbaren Matrix \mathbf{A} vgl. `ginv()` aus dem `MASS` Paket. Für solche Matrizen ermittelt `Null()` aus demselben Paket eine Basis des Kerns von \mathbf{A}^t (*null space*).

```

> a1 <- c(3, 4, 1, 8, 2)                      # Vektor
> sqrt(crossprod(a1))                         # seine Norm (euklidische Länge)
[1,] 9.69536

> sqrt(sum(a1^2))                            # Kontrolle ...
> a2 <- c(6, 9, 10, 8, 7)                     # weiterer Vektor
> A <- cbind(a1, a2)                          # verbinde Vektoren zu Matrix
> sqrt(diag(crossprod(A)))                   # Spaltennormen der Matrix
      a1         a2
9.69536  18.16590

> sqrt(colSums(A^2))                          # Kontrolle ...

```

`norm(<Matrix>, type=<Norm>)"` berechnet verschiedene Matrixnormen. Über das Argument `type` lässt sich der genaue Typ auswählen, die Frobenius-Norm erhält man etwa über `type="F"`. Fasst man eine $(p \times q)$ -Matrix A als $(p \cdot q)$ -Vektor a auf, ist die Frobenius-Norm von A gleich der Minkowski-2-Norm von a , also gleich dessen euklidischer Länge.

```

> norm(A, type="F")                           # Frobenius-Norm
[1] 20.59126

```

```

# Kontrolle: 2-Norm des zugehörigen Vektors
> sqrt(crossprod(c(A)))                     # ...

```

Der Abstand zwischen zwei Vektoren ist gleich der Norm des Differenzvektors. Mit `dist()` ist jedoch auch eine Funktion verfügbar, die unterschiedliche Abstandsmaße direkt berechnen kann.

```
> dist(x=<Matrix>, method=<Abstandsmaß>, diag=FALSE, upper=FALSE, p=2)
```

Das Argument `x` erwartet eine zeilenweise aus Koordinatenvektoren zusammengestellte Matrix. In der Voreinstellung werden alle paarweisen euklidischen Abstände zwischen ihren Zeilen berechnet. Über `method` lassen sich auch andere Abstandsmaße wählen, etwa die City-Block-Metrik mit "manhattan" oder die Minkowski- p -Norm mit "minkowski". Ein konkretes p kann in diesem Fall für `p` übergeben werden. Die Ausgabe enthält in der Voreinstellung die untere Dreiecksmatrix der paarweisen Abstände. Soll auch die Diagonale ausgegeben werden, ist `diag=TRUE` zu setzen, ebenso `upper=TRUE` für die obere Dreiecksmatrix.

```

> B <- matrix(sample(-20:20, 12, replace=TRUE), ncol=3)
> dist(B, diag=TRUE, upper=TRUE)
      1         2         3         4
1 0.000000 26.095977 29.698485 17.262677
2 26.095977 0.000000 7.681146 30.282008
3 29.698485 7.681146 0.000000 32.954514
4 17.262677 30.282008 32.954514 0.000000

```

```

# Kontrolle für den euklidischen Abstand der 1. und 2. Zeile
> sqrt(crossprod(B[1, ] - B[2, ]))
[1,] 26.09598

```

12.1.4 Mahalanobistransformation und Mahalanobisdistanz

Die Mahalanobistransformation ist eine affine Transformation einer multivariaten Variable, deren Zentroid in den Ursprung des Koordinatensystems verschoben und deren Kovarianzmatrix die zugehörige Einheitsmatrix wird. In diesem Sinne handelt es sich um eine Verallgemeinerung der z -Transformation univariater Variablen. Ist \mathbf{S} die Kovarianzmatrix einer multivariaten Variable X mit Zentroid $\bar{\mathbf{x}}$, ergibt $\mathbf{S}^{-\frac{1}{2}}(\mathbf{x} - \bar{\mathbf{x}})$ die Mahalanobistransformierte eines konkreten Datenvektors \mathbf{x} .⁸ Entsprechend ist $(\mathbf{S}^{-\frac{1}{2}}\dot{\mathbf{X}}^t)^t$ die Mahalanobistransformation mehrerer Datenvektoren, die zeilenweise die Matrix \mathbf{X} bilden, deren spaltenweise Zentrierte $\dot{\mathbf{X}}$ ist.

Im Beispiel seien an mehreren Bewerbern für eine Arbeitsstelle drei Eigenschaften erhoben worden. Zunächst wird `rmvnorm()` aus dem Paket `mvtnorm` verwendet, um Zufallsvektoren einer multinormalverteilten dreidimensionalen Variable zu simulieren. Die Verwendung von `rmvnorm()` gleicht der von `rnorm()`, lediglich muss hier das theoretische Zentroid μ für das Argument `mean` und die theoretische Kovarianzmatrix Σ für `sigma` angegeben werden. Die erzeugten Daten werden dann einer Mahalanobistransformation unterzogen (vgl. Abschn. 12.1.6 für die Berechnung von $\mathbf{S}^{-\frac{1}{2}}$ durch Diagonalisieren von \mathbf{S} mittels Spektralzerlegung durch `eigen()`).

```
# theoretische 3x3 Kovarianzmatrix
> sigma <- matrix(c(4,2,-3, 2,16,-1, -3,-1,9), byrow=TRUE, ncol=3)
> mu     <- c(-3, 2, 4)                      # theoretisches Zentroid
> N      <- 100                                # Anzahl Bewerber
> library(mvtnorm)                            # für rmvnorm()
> X      <- round(rmvnorm(N, mean=mu, sigma=sigma))    # Zufallsvektoren
> ctr   <- colMeans(X)                         # empirisches Zentroid
> S      <- cov(X)                            # empirische Kovarianzmatrix
> Seig  <- eigen(S)                           # Eigenwerte und -vektoren von S
> sqrtD <- sqrt(Seig$values)                  # Wurzel aus Eigenwerten

# berechne  $\mathbf{S}^{-\frac{1}{2}}$  durch Diagonalisieren
> SsqrtInv <- Seig$vectors %*% diag(1/sqrtD) %*% t(Seig$vectors)

# Differenzvektoren zwischen Daten und Zentroid
> Xdot <- scale(X, center=ctr, scale=FALSE)

# Mahalanobistransformation der Daten
> Xmt <- t(SsqrtInv %*% t(Xdot))

# Kontrolle: Kovarianzmatrix der transformierten Daten: Einheitsmatrix
> cov(Xmt)                                     # ...

# Kontrolle: Zentroid der transformierten Daten: Null-Vektor
> colMeans(Xmt)                               # ...
```

⁸Jede Transformation der Form $\mathbf{G}\mathbf{S}^{-\frac{1}{2}}(\mathbf{x} - \bar{\mathbf{x}})$ mit \mathbf{G} als Orthogonalmatrix ($\mathbf{G}^t = \mathbf{G}^{-1}$) würde ebenfalls eine multivariate z -Transformation liefern.

Die quadrierte Mahalanobisdistanz zwischen zwei Vektoren \mathbf{x} und \mathbf{y} bzgl. einer Kovarianzmatrix \mathbf{S} ist definiert als $(\mathbf{x} - \mathbf{y})^t \mathbf{S}^{-1} (\mathbf{x} - \mathbf{y})$ und lässt sich durch `mahalanobis()` berechnen.

```
> mahalanobis(x=<Matrix>, center=<Vektor>, cov=<Kovarianzmatrix>)
```

Das Argument `x` erwartet entweder einen Vektor oder eine zeilenweise aus Koordinatenvektoren zusammengestellte Matrix. Für `center` ist ein Vektor anzugeben, dessen jeweilige Distanz zu den Vektoren aus `x` berechnet wird. Die Kovarianzmatrix, bzgl. der die Transformation durchgeführt werden soll, ist für `cov` zu nennen. Häufig ist `center` das für die Mahalanobistransformation verwendete Zentroid der Variablen, von denen die Koordinatenvektoren in `x` stammen, und `cov` die Kovarianzmatrix dieser Variablen. Die Ausgabe ist ein Vektor mit den quadrierten Mahalanobis-Distanzen der Zeilen von `x` zum Vektor `center`.

Im obigen Beispiel sei der Bewerber zu bevorzugen, der einem Idealprofil, also vorher festgelegten konkreten Ausprägungen für jede Merkmalsdimension, am ehesten entspricht. Bei der Berechnung der Nähe zwischen Bewerber und Profil i.S. der Mahalanobisdistanz sind dabei Streuungen und Korrelationen der einzelnen Merkmale mit zu berücksichtigen.

```
> ideal <- c(1, 2, 3)                      # Idealprofil
> x      <- X[1, ]                         # Bewerber 1
> y      <- X[2, ]                         # Bewerber 2
> mat    <- rbind(x, y)                     # Bewerber zeilenweise als Matrix

# Quadrat der Mahalanobisdistanzen zum Idealprofil
> mahalanobis(mat, ideal, S)
      x          y
11.286151  1.529668

# manuelle Kontrolle
> Sinv <- solve(S)                      # Inverse der empirischen Kovarianzmatrix
> t(x-ideal) %*% Sinv %*% (x-ideal)     # quadrierte Distanz 1
[1,] 11.28615

> t(y-ideal) %*% Sinv %*% (y-ideal)      # quadrierte Distanz 2
[1,] 1.529668
```

Um den Bewerber mit der geringsten Mahalanobisdistanz zum Idealprofil unter allen Bewerbern zu identifizieren, kann auf `min()` und `which.min()` zurückgegriffen werden.

```
> mDist <- mahalanobis(X, ideal, S)        # quadr. Distanz alle Bewerber
> min(mDist)                                # geringste quadrierte Distanz
[1] 0.2095796

> (idxMin <- which.min(mDist))              # zugehöriger Bewerber
[1] 16

> X[idxMin, ]                                # sein Profil
[1] 1 1 2
```

Die Mahalanobisdistanz zweier Vektoren ist gleich deren euklidischer Distanz, nachdem beide derselben Mahalanobistransformation unterzogen wurden.

```
# Mahalanobistransformation des Idealprofils
> idealM <- t(SqrtInv %*% (ideal - ctr))

# Quadrat der euklidischen Distanz des transformierten ersten
# Bewerbers zum transformierten Idealprofil
> crossprod(Xmt[1, ] - t(idealM))
[1,] 11.28615

> crossprod(Xmt[2, ] - t(idealM))           # zweiter Bewerber
[1,] 1.529668
```

12.1.5 Kennwerte von Matrizen

Die Spur (*trace*) $\text{tr}(\mathbf{A}) = \sum_{i=1}^p a_{ii}$ einer $(p \times p)$ -Matrix \mathbf{A} erhält man mit `sum(diag(<Matrix>))`. Zudem sei an $\text{tr}(\mathbf{A}^t \mathbf{A}) = \text{tr}(\mathbf{A} \mathbf{A}^t)$ ebenso erinnert⁹ wie an $\text{tr}(\mathbf{AB}) = \text{tr}(\mathbf{BA})$ und $\text{tr}(\mathbf{A} + \mathbf{B}) = \text{tr}(\mathbf{B} + \mathbf{A})$, wenn auch \mathbf{B} eine $(p \times p)$ -Matrix ist.

```
> (A <- matrix(c(9, 1, 1, 4), nrow=2))          # Matrix A
      [,1]   [,2]
[1,]     9     1
[2,]     1     4

> sum(diag(A))                                # Spur von A
[1] 13

> sum(diag(t(A) %*% A))                      # Spur(A'A)
[1] 99

> sum(diag(A %*% t(A)))                      # Spur(AA')
[1] 99

> sum(A^2)                                     # Kontrolle: Summe quadr. Elemente
[1] 99
```

Für die Determinante $\det(\mathbf{A})$ einer quadratischen Matrix \mathbf{A} steht `det(<Matrix>)` zur Verfügung. Genau dann, wenn \mathbf{A} invertierbar ist, gilt $\det(\mathbf{A}) \neq 0$. Zudem ist $\det(\mathbf{A}) = \det(\mathbf{A}^t)$. Für die Determinante zweier $(p \times p)$ -Matrizen \mathbf{A} und \mathbf{B} gilt $\det(\mathbf{AB}) = \det(\mathbf{BA}) = \det(\mathbf{A}) \cdot \det(\mathbf{B})$. Für eine obere $(p \times p)$ -Dreiecksmatrix \mathbf{D} (insbesondere also für Diagonalmatrizen) ist $\det(\mathbf{D}) = \prod_{i=1}^p d_{ii}$, das Produkt der Diagonalelemente. Zusammengenommen folgt für invertierbare Matrizen $\det(\mathbf{A}^{-1}) = \frac{1}{\det(\mathbf{A})}$.¹⁰

⁹Es sei \mathbf{a}_i die i -te Zeile und $\mathbf{a}_{.j}$ die j -te Spalte von \mathbf{A} . Dann gilt $\text{tr}(\mathbf{A}^t \mathbf{A}) = \sum_j \mathbf{a}_{.j}^t \mathbf{a}_{.j} = \sum_j \sum_i a_{ij}^2 = \sum_i \sum_j a_{ij}^2 = \sum_i \mathbf{a}_i^t \mathbf{a}_i = \text{tr}(\mathbf{A} \mathbf{A}^t)$.

¹⁰ $\det(\mathbf{A}) \cdot \det(\mathbf{A}^{-1}) = \det(\mathbf{AA}^{-1}) = \det(\mathbf{I}) = \prod_{i=1}^p 1 = 1^p = 1$.

```

> det(A)                                # Determinante von A
[1] 35

# Determinante des Produkts = Produkt der Determinanten
> B <- matrix(c(-3, 4, -1, 7), nrow=2)
> all.equal(det(A %*% B), det(A) * det(B))
[1] TRUE

# Determinante einer Diagonalmatrix = Produkt der Diagonalelemente
> det(diag(1:4))
[1] 24

# Determinante der Inversen = Kehrwert der Determinante
> Ainv <- solve(A)                      # Inverse
> all.equal(1/det(A), det(Ainv))
[1] TRUE

```

Während der Rang einer Matrix als Maximum der Anzahl linear unabhängiger Zeilen bzw. Spalten klar definiert ist, wirft seine numerische Berechnung für beliebige Matrizen Probleme auf, weil Computer Zahlen nur mit endlicher Genauigkeit darstellen können (vgl. Abschn. 1.3.6). Über die QR-Zerlegung mit `qr(<Matrix>)` (vgl. Abschn. 12.1.6) lässt sich der Rang aber meist zuverlässig ermitteln. Er befindet sich in der ausgegebenen Liste in der Komponente `rank`. Eine $(p \times p)$ -Matrix \mathbf{X} ist genau dann invertierbar, wenn $\text{Rang}(\mathbf{X}) = p$ gilt.

```

> qrA <- qr(A)                          # QR-Zerlegung
> qrA$rank                               # Rang
[1] 2

```

Eigenwerte und -vektoren einer quadratischen Matrix berechnet `eigen(<Matrix>)`. Beide werden als Komponenten einer Liste ausgegeben – die Eigenwerte als Vektor in der Komponente `values`,¹¹ die normierten Eigenvektoren als Spalten einer Matrix in der Komponente `vectors`. Die Summe der Eigenwerte einer Matrix ist gleich ihrer Spur, das Produkt der Eigenwerte gleich der Determinante.

```

> (eigA <- eigen(A))                   # Eigenwerte und -vektoren von A
$values
[1] 9.192582 3.807418

$vectors
[,1]      [,2]
[1,] -0.9819564  0.1891075
[2,] -0.1891075 -0.9819564

# Matrix ist symmetrisch -> Matrix aus Eigenvektoren ist orthogonal

```

¹¹Eigenwerte werden entsprechend ihrer algebraischen Vielfachheit ggf. mehrfach aufgeführt. Auch Matrizen mit komplexen Eigenwerten sind zugelassen. Da in der Statistik vor allem Eigenwerte von Kovarianzmatrizen interessant sind, konzentriert sich die Darstellung hier auf den Fall reeller symmetrischer Matrizen. Ihre Eigenwerte sind alle reell, zudem stimmen algebraische und geometrische Vielfachheiten überein.

```
> zapsmall(eigA$vectors %*% t(eigA$vectors))
 [,1] [,2]
[1,]    1    0
[2,]    0    1

> sum(eigA$values)                      # Spur = Summe der Eigenwerte
[1] 13

> prod(eigA$values)                     # Determinante = Produkt der Eigenwerte
[1] 35
```

Die Kondition κ einer reellen Matrix \mathbf{X} ist für invertierbare \mathbf{X} gleich dem Produkt $\|\mathbf{X}\| \cdot \|\mathbf{X}^{-1}\|$ der Normen von \mathbf{X} und ihrer Inversen \mathbf{X}^{-1} , andernfalls gleich $\|\mathbf{X}\| \cdot \|\mathbf{X}^+\|$ mit \mathbf{X}^+ als Pseudoinverser. Das Ergebnis hängt damit von der Wahl der Matrixnorm ab, wobei die 2-Norm üblich ist. Die 2-Norm von \mathbf{X} ist gleich der Wurzel aus dem größten Eigenwert von $\mathbf{X}^t \mathbf{X}$, im Spezialfall symmetrischer Matrizen damit gleich dem betragsmäßig größten Eigenwert von \mathbf{X} . Zur Berechnung von κ dient `kappa(Matrix, exact=FALSE)`, wobei für eine numerisch aufwendigere, aber deutlich präzisere Bestimmung von κ das Argument `exact=TRUE` zu setzen ist.

```
> X <- matrix(c(20, 26, 10, 19, 29, 27, 20, 12, 17, 23, 27, 25), nrow=4)
> kappa(X, exact=TRUE)                         # Kondition kappa
[1] 7.931935

# Kontrolle: Produkt der 2-Norm jeweils von X und X+
> Xplus <- solve(t(X) %*% X) %*% t(X)          # Pseudoinverse X+
> norm(X, type="2") * norm(Xplus, type="2")       # Kondition kappa
[1] 7.931935
```

Bei Verwendung der 2-Norm gilt zudem $\kappa = \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}}$ mit λ_{\max} als größtem und λ_{\min} als kleinstem Eigenwert $\neq 0$ von $\mathbf{X}^t \mathbf{X}$.

```
> evX <- eigen(t(X) %*% X)$values            # Eigenwerte X^t X
> sqrt(max(evX) / min(evX[evX >= .Machine$double.eps]))
[1] 7.931935
```

Der Konditionsindex gibt für jeden Eigenwert λ_i den Quotienten $\sqrt{\frac{\lambda_i}{\lambda_{\min}}}$ an.

```
> sqrt(evX / min(evX[evX >= .Machine$double.eps]))
[1] 7.931935 1.502511 1.000000
```

12.1.6 Zerlegungen von Matrizen

Matrizen lassen sich auf unterschiedliche Weise *zerlegen*, also als Produkt anderer Matrizen darstellen. Diese Eigenschaft lässt sich in manchen Berechnungen ausnutzen, um eine höhere numerische Genauigkeit zu erreichen als bei der direkten Umsetzung mathematischer Formeln – etwa in der Regression (Bates, 2004).

Eine reelle symmetrische Matrix ist mit dem Spektralsatz diagonalisierbar, d. h. als *Spektralzerlegung* $\mathbf{G}\mathbf{D}\mathbf{G}^t$ darstellbar. Dabei ist \mathbf{G} die Orthogonalmatrix ($\mathbf{G}^t = \mathbf{G}^{-1}$) mit den normierten Eigenvektoren in den Spalten und \mathbf{D} die aus den Eigenwerten in zugehöriger Reihenfolge gebildete reelle Diagonalmatrix.

```
> X      <- matrix(c(20, 26, 10, 19, 29, 27, 20, 12, 17, 23, 27, 25), nrow=4)
> S      <- cov(X)                                # Kovarianzmatrix S von X (symmetrisch)
> eigS <- eigen(S)                               # Eigenwerte und -vektoren von S
> G      <- eigS$vectors                        # Matrix der Eigenvektoren
> D      <- diag(eigS$values)                   # Diagonalmatrix aus Eigenwerten
> all.equal(S, G %*% D %*% t(G))   # Spektralzerlegung S = G D G^t
[1] TRUE
```

Ist eine Matrix \mathbf{X} diagonalisierbar, lassen sich Matrizen finden, durch die \mathbf{X} ebenfalls berechnet werden kann, was in verschiedenen Anwendungsfällen nützlich ist. So lässt sich \mathbf{X} dann als Quadrat einer Matrix $\mathbf{A} = \mathbf{G}\mathbf{D}^{\frac{1}{2}}\mathbf{G}^t$ darstellen ($\mathbf{X} = \mathbf{A}^2 = \mathbf{A}\mathbf{A}$),¹² oder als Produkt einer Matrix \mathbf{N} mit deren Transponierter ($\mathbf{X} = \mathbf{N}\mathbf{N}^t$). Dabei ist $\mathbf{N} = \mathbf{G}\mathbf{D}^{\frac{1}{2}}$ die spaltenweise aus den Eigenvektoren zusammengestellte Matrix \mathbf{X} , deren Spaltenlängen jeweils gleich der Wurzel aus den zugehörigen Eigenwerten sind.¹³

```
# stelle S als Quadrat A^2 einer Matrix A dar
> sqrtD <- diag(sqrt(eigS$values))    # Diagonalmatrix Wurzel der Eigenwerte
> A      <- G %*% sqrtD %*% t(G)      # Matrix A = G D^(1/2) G^t
> all.equal(S, A %*% A)                  # Zerlegung S = A^2
[1] TRUE

# stelle S als Produkt N N^t dar
> N <- eigS$vectors %*% sqrt(diag(eigS$values))          # Matrix N
> all.equal(S, N %*% t(N))           # Zerlegung S = N N^t
[1] TRUE
```

Die Singulärwertzerlegung einer beliebigen $(n \times p)$ -Matrix \mathbf{X} liefert `svd(Matrix)` in Form einer Liste. Deren Komponente \mathbf{d} ist der Vektor der Singulärwerte, \mathbf{u} die spaltenweise aus den Links-Singulärvektoren und \mathbf{v} die spaltenweise aus den Rechts-Singulärvektoren zusammengesetzte Matrix. Insgesamt gilt damit $\mathbf{X} = \mathbf{u}\mathbf{D}\mathbf{v}^t$, wenn \mathbf{D} die aus den Singulärwerten gebildete Diagonalmatrix ist. Die Spalten von \mathbf{u} sind Eigenvektoren von $\mathbf{X}\mathbf{X}^t$, die von \mathbf{v} Eigenvektoren von $\mathbf{X}^t\mathbf{X}$. Die Singulärwerte von \mathbf{X} sind gleich der Wurzel aus den Eigenwerten von $\mathbf{X}^t\mathbf{X}$. Die Kondition κ (vgl. Abschn. 12.1.5) ist also auch gleich dem Quotienten vom größten zum kleinsten Singulärwert $\neq 0$ von \mathbf{X} .

```
> svdX <- svd(X)                            # Singulärwertzerlegung von X
> all.equal(X, svdX$u %*% diag(svdX$d) %*% t(svdX$v))  # X = u D v^t
[1] TRUE

# Wurzel aus Eigenwerten von X^t X = Singulärwerte von X
> all.equal(sqrt(eigen(t(X) %*% X)$values), svdX$d)
```

¹² $\mathbf{A}\mathbf{A} = \mathbf{G}\mathbf{D}^{\frac{1}{2}}\mathbf{G}^t\mathbf{G}\mathbf{D}^{\frac{1}{2}}\mathbf{G}^t = \mathbf{G}\mathbf{D}^{\frac{1}{2}}\mathbf{D}^{\frac{1}{2}}\mathbf{G}^t = \mathbf{G}\mathbf{D}\mathbf{G}^t = \mathbf{X}$. Siehe auch `sqrtm()` aus dem Paket `expm`.

¹³ $\mathbf{N}\mathbf{N}^t = \mathbf{G}\mathbf{D}^{\frac{1}{2}}(\mathbf{G}\mathbf{D}^{\frac{1}{2}})^t = \mathbf{G}\mathbf{D}^{\frac{1}{2}}(\mathbf{D}^{\frac{1}{2}})^t\mathbf{G}^t = \mathbf{G}\mathbf{D}\mathbf{G}^t = \mathbf{X}$.

[1] TRUE

Die mit `chol(<Matrix>)` durchgeführte Cholesky-Zerlegung einer symmetrischen, positiv-definiten Matrix \mathbf{X} berechnet eine obere Dreiecksmatrix \mathbf{R} , so dass $\mathbf{X} = \mathbf{R}^t \mathbf{R}$ gilt.

```
> R <- chol(S)                      # Cholesky-Zerlegung von S
> all.equal(S, t(R) %*% R)           # Zerlegung S = R^t R
[1] TRUE
```

Die QR -Zerlegung einer beliebigen Matrix \mathbf{X} erhält man mit der `qr(<Matrix>)` Funktion. Sie berechnet Matrizen \mathbf{Q} und \mathbf{R} , so dass $\mathbf{X} = \mathbf{Q}\mathbf{R}$ gilt, wobei \mathbf{Q} eine Orthogonalmatrix derselben Dimensionierung wie \mathbf{X} ist ($\mathbf{Q}^t = \mathbf{Q}^{-1}$) und \mathbf{R} eine obere Dreiecksmatrix. Die Ausgabe ist eine Liste, die den Rang von \mathbf{X} in der Komponente `rank` enthält. Um die Matrizen \mathbf{Q} und \mathbf{R} zu erhalten, müssen die Hilfsfunktionen `qr.Q(<QR-Liste>)` bzw. `qr.R(<QR-Liste>)` auf die von `qr()` ausgegebene Liste angewendet werden.

```
> qrX <- qr(X)                      # QR-Zerlegung von X
> Q   <- qr.Q(qrX)                  # extrahiere Q
> R   <- qr.R(qrX)                  # extrahiere R
> all.equal(X, Q %*% R)              # Zerlegung X = Q R
[1] TRUE
```

12.1.7 Orthogonale Projektion

In vielen statistischen Verfahren spielt die orthogonale Projektion von Daten im durch n Beobachtungsobjekte aufgespannten n -dimensionalen Personenraum U auf bestimmte Unterräume eine entscheidende Rolle. Hier sei V ein solcher p -dimensionaler Unterraum ($n > p$) und die $(n \times p)$ -Matrix \mathbf{A} spaltenweise eine Basis von V ($\text{Rang}(\mathbf{A}) = p$). Für jeden Datensatz \mathbf{x} aus U liefert die orthogonale Projektion den Punkt \mathbf{v} aus V mit dem geringsten euklidischen Abstand zu \mathbf{x} (Abb. 12.1). \mathbf{v} ist dadurch eindeutig bestimmt, dass $(\mathbf{x} - \mathbf{v}) \perp V$ gilt, $\mathbf{x} - \mathbf{v}$ also senkrecht auf V und damit senkrecht auf den Spalten von \mathbf{A} steht ($\mathbf{A}^t(\mathbf{x} - \mathbf{v}) = \mathbf{0}$). Ist \mathbf{y} der Koordinatenvektor von \mathbf{v} bzgl. der Basis \mathbf{A} , erfüllt $\mathbf{v} = \mathbf{Ay}$ somit die Bedingung $\mathbf{A}^t(\mathbf{x} - \mathbf{Ay}) = \mathbf{0}$, was sich zu $\mathbf{A}^t \mathbf{Ay} = \mathbf{A}^t \mathbf{x}$ umformen lässt – den *Normalengleichungen*.

Da \mathbf{A} als Basis vollen Spaltenrang besitzt, existiert die Inverse $(\mathbf{A}^t \mathbf{A})^{-1}$, weshalb der Koordinatenvektor \mathbf{y} des orthogonal auf V projizierten Vektors \mathbf{x} bzgl. der Basis \mathbf{A} durch $(\mathbf{A}^t \mathbf{A})^{-1} \mathbf{A}^t \mathbf{x}$ gegeben ist.¹⁴ Die Koordinaten bzgl. der Standardbasis berechnen sich entsprechend durch $\mathbf{A}(\mathbf{A}^t \mathbf{A})^{-1} \mathbf{A}^t \mathbf{x}$. Die Matrix $\mathbf{A}(\mathbf{A}^t \mathbf{A})^{-1} \mathbf{A}^t$ wird als orthogonale Projektion \mathbf{P} auf V bezeichnet.¹⁵ Ist \mathbf{C} eine Orthogonalbasis von V ($\mathbf{C}^t \mathbf{C} = \mathbf{I}$ mit \mathbf{I} als $(p \times p)$ -Einheitsmatrix), gilt $\mathbf{P} = \mathbf{C}(\mathbf{C}^t \mathbf{C})^{-1} \mathbf{C}^t = \mathbf{CC}^t$.

¹⁴ $(\mathbf{A}^t \mathbf{A})^{-1} \mathbf{A}^t$ ist die Pseudoinverse \mathbf{A}^+ von \mathbf{A} .

¹⁵ Im Kontext linearer Modelle ist \mathbf{P} die *Hat-Matrix* (vgl. Abschn. 6.3.1).

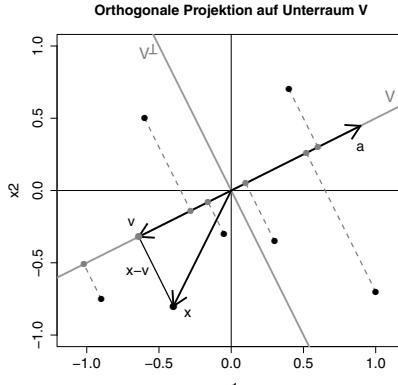


Abbildung 12.1: Orthogonale Projektion von Vektoren \mathbf{x} auf Unterraum V mit Basis \mathbf{a}

Eigenschaften

Im Fall eines eindimensionalen Unterraums V , dessen Basisvektor \mathbf{a} bereits normiert ist ($\|\mathbf{a}\|^2 = \mathbf{a}^t \mathbf{a} = 1$), vereinfacht sich die Projektion zu $\mathbf{a} \mathbf{a}^t \mathbf{x}$, die Koordinaten des projizierten Vektors bzgl. \mathbf{a} liefert entsprechend $\mathbf{a}^t \mathbf{x}$ – das Skalarprodukt von \mathbf{a} und \mathbf{x} .

Vektoren, die bereits aus V stammen, bleiben durch die Projektion unverändert, insbesondere ist also $\mathbf{P}\mathbf{A} = \mathbf{A}$.¹⁶ Orthogonale Projektionen sind idempotent ($\mathbf{P}^2 = \mathbf{P}$) und symmetrisch ($\mathbf{P}^t = \mathbf{P}$),¹⁷ was sie auch bereits vollständig charakterisiert. Ferner gilt $\text{Rang}(\mathbf{P}) = \text{Rang}(\mathbf{A}) = p$, der Dimension von V . Zudem ist $\text{Rang}(\mathbf{P}) = \text{tr}(\mathbf{P})$.¹⁸ \mathbf{P} besitzt nur die Eigenwerte 1 mit Vielfachheit p und 0 mit Vielfachheit $n - p$. Die Eigenvektoren zum Eigenwert 1 bilden dabei eine Orthogonalbasis von V , da Vektoren \mathbf{v} aus V durch \mathbf{P} unverändert bleiben. Ist für zwei orthogonale Projektionen \mathbf{P}_1 und \mathbf{P}_2 ihr Produkt $\mathbf{P}_1\mathbf{P}_2$ symmetrisch, gilt $\mathbf{P}_1\mathbf{P}_2 = \mathbf{P}_2\mathbf{P}_1$.¹⁹

Die Projektion auf das orthogonale Komplement V^\perp von V berechnet sich mit der $(n \times n)$ -Einheitsmatrix \mathbf{I} als $\mathbf{I} - \mathbf{P}$.²⁰ Der Rang von $\mathbf{I} - \mathbf{P}$ ist gleich $n - \text{Rang}(\mathbf{A})$, der Dimension von V^\perp . Für \mathbf{v} aus V gilt $(\mathbf{I} - \mathbf{P})\mathbf{v} = \mathbf{0}$,²¹ insbesondere ist also $(\mathbf{I} - \mathbf{P})\mathbf{A} = \mathbf{0}$.

Beispiele

Als Beispiel sollen die im vorangehenden Abschnitt simulierten Daten mit der Projektion \mathbf{P}_1 im durch die Beobachtungsobjekte aufgespannten n -dimensionalen Personenraum auf den

¹⁶Ist \mathbf{v} aus V , lässt sich $\mathbf{v} = \mathbf{C}\mathbf{y}$ schreiben, wobei \mathbf{y} der Koordinatenvektor von \mathbf{v} bzgl. einer Orthogonalbasis \mathbf{C} von V ist. Damit folgt $\mathbf{P}\mathbf{v} = \mathbf{C}\mathbf{C}^t\mathbf{v} = \mathbf{C}\mathbf{C}^t\mathbf{C}\mathbf{y} = \mathbf{C}\mathbf{y} = \mathbf{v}$.

¹⁷Zunächst gilt $\mathbf{P}^2 = (\mathbf{C}\mathbf{C}^t)(\mathbf{C}\mathbf{C}^t) = \mathbf{C}\mathbf{C}^t = \mathbf{P}$. Weiter gilt $\mathbf{P}^t = (\mathbf{C}\mathbf{C}^t)^t = \mathbf{C}\mathbf{C}^t = \mathbf{P}$.

¹⁸ $\text{tr}(\mathbf{P}) = \text{tr}(\mathbf{C}\mathbf{C}^t) = \text{tr}(\mathbf{C}^t\mathbf{C}) = \text{tr}(\mathbf{I}) = p$.

¹⁹ $\mathbf{P}_1\mathbf{P}_2 = (\mathbf{P}_1\mathbf{P}_2)^t = \mathbf{P}_2^t\mathbf{P}_1^t = \mathbf{P}_2\mathbf{P}_1$.

²⁰Ist $\mathbf{w} = (\mathbf{I} - \mathbf{P})\mathbf{x}$ aus V^\perp und \mathbf{v} aus V , muss $\mathbf{w}^t \mathbf{v} = \mathbf{0}$ gelten. \mathbf{v} lässt sich als $\mathbf{v} = \mathbf{C}\mathbf{y}$ schreiben, wobei \mathbf{y} der Koordinatenvektor von \mathbf{v} bzgl. einer Orthogonalbasis \mathbf{C} von V ist. Nun gilt $\mathbf{w}^t \mathbf{v} = ((\mathbf{I} - \mathbf{P})\mathbf{x})^t \mathbf{C}\mathbf{y} = \mathbf{x}^t (\mathbf{I} - \mathbf{P})\mathbf{C}\mathbf{y} = (\mathbf{x}^t - \mathbf{x}^t \mathbf{C}\mathbf{C}^t)\mathbf{C}\mathbf{y} = \mathbf{x}^t \mathbf{C}\mathbf{y} - \mathbf{x}^t \mathbf{C}\mathbf{C}^t \mathbf{C}\mathbf{y} = \mathbf{x}^t \mathbf{C}\mathbf{y} - \mathbf{x}^t \mathbf{C}\mathbf{y} = \mathbf{0}$.

²¹ $(\mathbf{I} - \mathbf{P})\mathbf{v} = \mathbf{I}\mathbf{v} - \mathbf{P}\mathbf{v} = \mathbf{v} - \mathbf{v} = \mathbf{0}$, da \mathbf{v} durch \mathbf{P} unverändert bleibt.

eindimensionalen Unterraum projiziert werden, dessen Basisvektor $\mathbf{1}$ aus lauter 1-Einträgen besteht. Dies bewirkt, dass alle Werte jeweils durch den Mittelwert der zugehörigen Variable ersetzt werden.²² Die Projektion auf das orthogonale Komplement liefert die Differenzen zum zugehörigen Spaltenmittelwert. Damit ist $\mathbf{I} - \mathbf{P}_1$ gleich der Zentriermatrix (vgl. Abschn. 12.1.1).

```
> X <- matrix(c(20, 26, 10, 19, 29, 27, 20, 12, 17, 23, 27, 25), nrow=4)

# Basisvektor eines eindimensionalen Unterraums: lauter 1-Einträge
> ones <- rep(1, nrow(X))
> P1 <- ones %*% solve(t(ones) %*% ones) %*% t(ones)      # Projektion
> P1x <- P1 %*% X           # Koordinaten der Projektion
> head(P1x, n=3)           # Werte ersetzt durch Variablen-Mittelwerte
   [,1] [,2] [,3]
[1,] 18.75 22    23
[2,] 18.75 22    23
[3,] 18.75 22    23

> colMeans(X)             # Kontrolle: Spaltenmittel der Daten
[1] 18.75 22.00 23.00

# orthogonale Projektion auf normierten Vektor als Basis des Unterraums
> a <- ones / sqrt(crossprod(ones))          # normiere Basisvektor
> P2 <- a %*% t(a)                         # orthogonale Projektion
> all.equal(P1, P2)                          # Kontrolle: Vergleich mit erster Projektion
[1] TRUE

# Projektion auf orthogonales Komplement von P1
> IP1 <- diag(nrow(X)) - P1
> IP1x <- IP1 %*% X           # zentrierte Daten

# Kontrolle
> all.equal(IP1x, scale(X, center=colMeans(X), scale=FALSE),
+            check.attributes=FALSE)
[1] TRUE
```

Als weiteres Beispiel sollen die Daten im durch die Variablen aufgespannten dreidimensionalen Raum auf den zweidimensionalen Unterraum projiziert werden, dessen Basisvektoren die zwei ersten Vektoren der Standardbasis sind. Dies bewirkt, dass die dritte Komponente jedes Zeilenvektors der Datenmatrix auf 0 gesetzt wird.

```
# Basis eines zweidimensionalen Unterraums: erste 2 Standard-Basisvektoren
> A <- cbind(c(1, 0, 0), c(0, 1, 0))
> P3 <- A %*% solve(t(A) %*% A) %*% t(A)      # orthogonale Projektion
> Px3 <- t(P3 %*% t(X))                      # Koordinaten der Projektion
> head(Px3, n=3)                               # 3. Komponente auf 0 gesetzt
```

²² $\mathbf{P}_1 = \mathbf{1}_n(\mathbf{1}_n^t \mathbf{1}_n)^{-1} \mathbf{1}_n^t = \frac{1}{n} \mathbf{1}_n \mathbf{1}_n^t = \frac{1}{n} \mathbf{1}_{n \times n}.$

```
[,1] [,2] [,3]
[1,] 20    29    0
[2,] 26    27    0
[3,] 10    20    0
```

Mit Hilfe der QR -Zerlegung lassen sich orthogonale Projektionen numerisch stabiler berechnen als mit der direkten Umsetzung der mathematischen Formel: Mit $\mathbf{X} = \mathbf{Q}\mathbf{R}$ folgt für die Pseudoinverse $\mathbf{X}^+ = \mathbf{R}^{-1}\mathbf{Q}^t$ ²³ und für die Projektion (Hat-Matrix \mathbf{H}) $\mathbf{P} = \mathbf{Q}\mathbf{Q}^t$.²⁴

```
> qrX <- qr(X)                                # QR-Zerlegung
> Q     <- qr.Q(qrX)                           # extrahiere Q
> R     <- qr.R(qrX)                           # extrahiere R
> Xplus <- solve(t(X) %*% X) %*% t(X)        # Pseudoinverse X+
> all.equal(Xplus, solve(R) %*% t(Q))          # X+ = R^(-1) Q^t
[1] TRUE

> all.equal(X %*% Xplus, Q %*% t(Q))           # P = Q Q^t
[1] TRUE
```

12.2 Hauptkomponentenanalyse

Die Hauptkomponentenanalyse dient dazu, die Hauptstreuungsrichtungen multivariater Daten im durch die Variablen aufgespannten Raum zu identifizieren. Hauptkomponenten sind neue Variablen, die als Linearkombinationen der beobachteten Variablen gebildet werden und folgende Eigenschaften besitzen:

- Die Linearkombinationen sind standardisiert, d. h. die Koeffizientenvektoren haben jeweils die Länge 1. Die Koeffizientenvektoren sind die normierten Eigenvektoren der Kovarianzmatrix der Daten (bis auf potentiell unterschiedliche Vorzeichen i. S. von Rotationen um 180°) und damit paarweise orthogonal.²⁵
- Die Hauptkomponenten sind zentriert und paarweise unkorreliert.²⁶

²³Hier sei voller Spaltenrang von \mathbf{X} vorausgesetzt. Dann ist $\mathbf{X}^+ = (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{X}^t = ((\mathbf{Q}\mathbf{R})^t \mathbf{Q}\mathbf{R})^{-1} (\mathbf{Q}\mathbf{R})^t = (\mathbf{R}^t \mathbf{Q}^t \mathbf{Q}\mathbf{R})^{-1} \mathbf{R}^t \mathbf{Q}^t = (\mathbf{R}^t \mathbf{R})^{-1} \mathbf{R}^t \mathbf{Q}^t = \mathbf{R}^{-1} \mathbf{R}^{t-1} \mathbf{R}^t \mathbf{Q}^t = \mathbf{R}^{-1} \mathbf{Q}^t$.

²⁴ $\mathbf{X}\mathbf{X}^+ = \mathbf{Q}\mathbf{R}\mathbf{R}^{-1}\mathbf{Q}^t = \mathbf{Q}\mathbf{Q}^t$.

²⁵Es sei $\bar{\mathbf{x}}$ das Zentroid der spaltenweise aus den Variablen zusammengestellten Datenmatrix \mathbf{X} , \mathbf{x} ein Datenvektor und \mathbf{G} die Matrix der spaltenweise zusammengestellten normierten Eigenvektoren der Kovarianzmatrix \mathbf{S} von \mathbf{X} (vgl. Abschn. 12.1.5). Mit dem Spektralsatz ist \mathbf{G} eine Orthogonalmatrix ($\mathbf{G}^t = \mathbf{G}^{-1}$, vgl. Abschn. 12.1.6). Dann berechnet sich der zugehörige Vektor \mathbf{y} der Hauptkomponenten als $\mathbf{y} = \mathbf{G}^t(\mathbf{x} - \bar{\mathbf{x}})$.

²⁶Genauer gesagt ist ihre Kovarianz 0 – da ihre Varianz auch 0 sein kann, ist die Korrelation nicht immer definiert. \mathbf{S} lässt sich mit $\mathbf{S} = \mathbf{G}\mathbf{D}\mathbf{G}^t$ diagonalisieren (vgl. Abschn. 12.1.6). Dabei ist \mathbf{D} die zu \mathbf{G} gehörende, aus den Eigenwerten von \mathbf{S} gebildete Diagonalmatrix. Damit gilt $V(\mathbf{y}) = V(\mathbf{G}^t(\mathbf{x} - \bar{\mathbf{x}})) = V(\mathbf{G}^t\mathbf{x} - \mathbf{G}^t\bar{\mathbf{x}}) = V(\mathbf{G}^t\mathbf{x}) = \mathbf{G}^tV(\mathbf{x})\mathbf{G} = \mathbf{G}^t\mathbf{S}\mathbf{G} = \mathbf{G}^t\mathbf{G}\mathbf{D}\mathbf{G}^t\mathbf{G} = \mathbf{D}$. Die Varianzen der Hauptkomponenten sind also gleich den Eigenwerten der Kovarianzmatrix der Daten, die Kovarianzen sind 0.

- Der euklidische Abstand zwischen zwei Punkten im Raum der Hauptkomponenten ist derselbe wie im Raum der beobachteten Variablen.²⁷
- Die Streuung der Hauptkomponenten ist im folgenden Sinn sukzessive maximal: Die erste Hauptkomponente ist diejenige unter allen standardisierten Linearkombinationen mit der größten Streuung. Unter allen standardisierten Linearkombinationen, die mit der ersten Hauptkomponente unkorreliert sind, ist die zweite Hauptkomponente dann wieder diejenige mit der größten Streuung. Für die weiteren Hauptkomponenten gilt dies analog.

12.2.1 Berechnung

Für die Berechnung der Hauptkomponenten samt ihrer jeweiligen Streuung stehen `prcomp()` und `princomp()` zur Verfügung.

```
> prcomp(Matrix oder Modellformel), scale=FALSE, data=<Datensatz>
> princomp(Matrix oder Modellformel), data=<Datensatz>,
+           cor=FALSE, covmat)
```

In der ersten Variante akzeptieren bei Funktionen die spaltenweise aus den Variablen zusammengestellte Datenmatrix. Damit die Hauptkomponenten aus den standardisierten Variablen berechnet werden, ist `prcomp(..., scale=TRUE)` bzw. `princomp(..., cor=TRUE)` zu setzen. Alternativ lassen sich die Variablen in beiden Funktionen als rechte Seite einer Modellformel nennen. Stammen sie aus einem Datensatz, muss dieser unter `data` übergeben werden. `princomp()` erlaubt es zusätzlich, die Kovarianzmatrix der Daten über das Argument `covmat` separat zu spezifizieren. Dies könnte etwa für robuste Schätzungen der theoretischen Kovarianzmatrix genutzt werden (vgl. Abschn. 2.7.9).²⁸

```
# 2x2 Kovarianzmatrix für simulierte Zufallsvektoren
> sigma <- matrix(c(4, 2, 2, 3), ncol=2)
> mu     <- c(1, 2)                      # Zentroid für Zufallsvektoren
> N      <- 50                          # Anzahl Personen
> library(mvtnorm)                     # für rmvnorm()
> X      <- rmvnorm(N, mean=mu, sigma=sigma) # Multinormalverteilung
> (pca <- prcomp(X))                  # Hauptkomponentenanalyse
Standard deviations:
[1] 2.436560 1.270408
```

Rotation:

PC1	PC2
[1,] -0.7335078	0.6796810
[2,] -0.6796810	-0.7335078

Die Ausgabe von `prcomp()` ist eine Liste mit zwei Komponenten: die erste enthält den Vektor der korrigierten Streuungen der Hauptkomponenten (Überschrift `Standard deviations`). Dies

²⁷Es seien \mathbf{x}_1 und \mathbf{x}_2 zwei Datenvektoren mit zugehörigen Hauptkomponenten-Vektoren \mathbf{y}_1 und \mathbf{y}_2 . Dann gilt

$$\|\mathbf{x}_2 - \mathbf{x}_1\|^2 = (\mathbf{x}_2 - \mathbf{x}_1)^t(\mathbf{x}_2 - \mathbf{x}_1) = (\mathbf{x}_2 - \mathbf{x}_1)^t \mathbf{G} \mathbf{G}^t (\mathbf{x}_2 - \mathbf{x}_1) = (\mathbf{G}^t (\mathbf{x}_2 - \mathbf{x}_1))^t \mathbf{G}^t (\mathbf{x}_2 - \mathbf{x}_1) = \|\mathbf{G}^t (\mathbf{x}_2 - \mathbf{x}_1)\|^2 =$$

$$\|\mathbf{G}^t \mathbf{x}_2 - \mathbf{G}^t \mathbf{x}_1\|^2 = \|\mathbf{G}^t (\mathbf{x}_2 - \bar{\mathbf{x}}) - \mathbf{G}^t (\mathbf{x}_1 - \bar{\mathbf{x}})\|^2 = \|\mathbf{y}_2 - \mathbf{y}_1\|^2.$$

²⁸Für die robuste Hauptkomponentenanalyse vgl. das Paket `pcaPP` (Filzmoser, Fritz & Kalcher, 2014).

sind gleichzeitig die Wurzeln aus den Eigenwerten der korrigierten Kovarianzmatrix der Daten (vgl. Abschn. 12.1.5). Die zweite Komponente beinhaltet die als Spalten einer Matrix G zusammengestellten Koeffizienten der Linearkombinationen zur Bildung der Hauptkomponenten (Überschrift *Rotation*). Dies sind gleichzeitig die normierten Eigenvektoren der korrigierten Kovarianzmatrix.

```
> eig      <- eigen(cov(X))           # Eigenwerte, -vektoren Kovarianzmatrix
> eigVal <- eig$values            # Eigenwerte
> sqrt(eigVal)                  # deren Wurzel = Streuungen der HK
[1] 2.436560 1.270408

# normierte Eigenvektoren = Koeffizientenvektoren der Linearkombinationen
> (G <- eig$vectors)
[,1]          [,2]
[1,] -0.7335078  0.6796810
[2,] -0.6796810 -0.7335078
```

Die Hauptkomponenten kann man in einem Koordinatensystem ablesen, das seinen Ursprung im Zentroid der Daten hat. Die Achsen weisen in Richtung der Koeffizientenvektoren und haben dieselbe Einheit wie das Standard-Koordinatensystem (Abb. 12.2). Die Hauptkomponenten sind die orthogonale Projektion der zentrierten Daten auf die Geraden in Richtung der Eigenvektoren der Kovarianzmatrix (vgl. Abschn. 12.1.7). Die Streuung der projizierten Daten entlang der Geraden ist daher gleich der Streuung der zugehörigen Hauptkomponente. Sie ist zudem gleich der Wurzel aus dem entsprechenden Eigenwert der Kovarianzmatrix.

```
# normierte Eigenvektoren * Wurzel aus zugehörigem Eigenwert
> B      <- G %*% diag(sqrt(eigVal))
> ctr   <- colMeans(X)                # Zentroid
> xMat <- rbind(ctr[1] - B[1, ], ctr[1])
> yMat <- rbind(ctr[2] - B[2, ], ctr[2])

# Punkt-Steigungsform der durch Eigenvektoren definierten Achsen
> ab1 <- solve(cbind(1, xMat[, 1]))    # Achse 1
> ab2 <- solve(cbind(1, xMat[, 2]))    # Achse 2

# Daten im Streudiagramm darstellen
> plot(X, xlab="x", ylab="y", pch=20, asp=1,
+       main="Datenwolke und Hauptkomponenten")

> abline(coef=ab1, lwd=2, col="gray")      # Achse 1
> abline(coef=ab2, lwd=2, col="gray")      # Achse 2
> matlines(xMat, yMat, lty=1, lwd=6, col="blue")  # Streuungen HK
> points(ctr[1], ctr[2], pch=16, col="red", cex=3) # Zentroid

# Legende einfügen
> legend(x="topleft", legend=c("Daten", "Achsen HK",
+                               "Streuungen HK", "Zentroid"), pch=c(20, NA, NA, 16),
+                               lty=c(NA, 1, 1, NA), col=c("black", "gray", "blue", "red"))
```

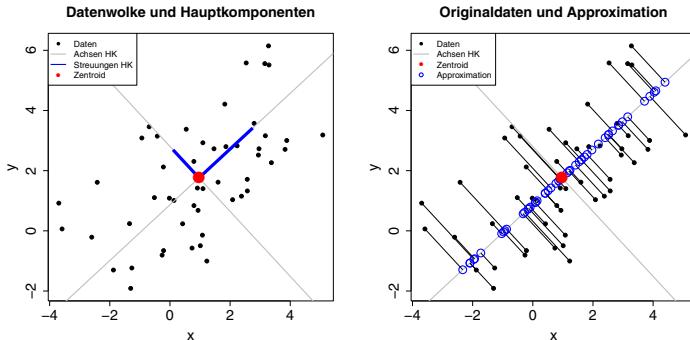


Abbildung 12.2: Hauptkomponentenanalyse: Koordinatensystem mit Achsen in Richtung der ins Zentroid verschobenen Eigenvektoren sowie Streuungen der Hauptkomponenten. Originaldaten und Approximation durch erste Hauptkomponente

Aus der Matrix \mathbf{G} der Koeffizienten der standardisierten Linearkombinationen und der spaltenweise zentrierten Datenmatrix $\dot{\mathbf{X}}$ berechnen sich die Hauptkomponenten als neue Variablen durch $\dot{\mathbf{X}}\mathbf{G}$. Man erhält sie mit `predict(prcomp-Objekt)`). Dabei lässt sich für das Argument `newdata` eine Matrix mit neuen Werten der ursprünglichen Variablen angeben, die dann in Werte auf den Hauptkomponenten umgerechnet werden.

```
> Xdot <- scale(X, center=TRUE, scale=FALSE)           # zentrierte Daten
> Y      <- Xdot %*% G                               # Hauptkomponenten
> head(Y, n=3)
   PC1          PC2
[1,] -4.257878 -1.171098
[2,] -1.021898 -0.370420
[3,]  1.500310  1.683688

> (Ysd <- apply(Y, 2, sd))                         # deren Streuungen
   PC1          PC2
2.436560  1.270408

> head(predict(pca), n=3)                           # Kontrolle mit prcomp()
   PC1          PC2
[1,] -4.257878 -1.171098
[2,] -1.021898 -0.370420
[3,]  1.500310  1.683688

# Hauptkomponenten für neue Daten berechnen
> Xnew <- matrix(1:4, ncol=2)
> predict(pca, newdata=Xnew)
   PC1          PC2
[1,] -0.8672799 -0.8858317
[2,] -2.2804688 -0.9396585
```

Die Streuungen der Hauptkomponenten, den Anteil ihrer Varianz an der Gesamtvarianz (i.S. der Spur der Kovarianzmatrix der Daten) sowie den kumulativen Anteil der Varianz der Hauptkomponenten an der Gesamtvarianz gibt `summary(PCA)` aus. Dabei ist `PCA` das Ergebnis einer Hauptkomponentenanalyse mit `prcomp()` oder `princomp()`.

```
> summary(pca)
Importance of components:
              PC1        PC2
Standard deviation   2.4366  1.2704
Proportion of Variance 0.7863  0.2137
Cumulative Proportion 0.7863  1.0000

# Kontrolle: Anteil der Varianz der HK an Gesamtvarianz
> Ysd^2 / sum(diag(cov(X)))
[1] 0.7862549 0.2137451
```

12.2.2 Dimensionsreduktion

Häufig dient die Hauptkomponentenanalyse der Datenreduktion: Wenn von Beobachtungsobjekten Werte von sehr vielen Variablen vorliegen, ist es oft wünschenswert, die Daten durch Werte von weniger Variablen möglichst gut zu approximieren. Als Kriterium dient dabei die Summe der quadrierten Abstände zwischen Originaldaten und Approximation. Eine perfekte Reproduktion der Originaldaten lässt sich zunächst wie folgt erreichen: Sei die Matrix \mathbf{B} spaltenweise aus den Koeffizientenvektoren der Hauptkomponenten zusammengestellt. Zusätzlich sollen die Spalten von \mathbf{B} als Länge die Streuung der jeweils zugehörigen Hauptkomponente besitzen, was mit $\mathbf{B} = \mathbf{G}\mathbf{D}^{\frac{1}{2}}$ erreicht wird. Weiter sei \mathbf{H} die Matrix der standardisierten Hauptkomponenten, die jeweils Mittelwert 0 und Varianz 1 besitzen. Ferner sei \bar{x} das Zentroid der Daten. Für die Datenmatrix \mathbf{X} gilt dann $\mathbf{X} = \mathbf{HB}^t + \bar{x}$.²⁹

```
> B <- G %*% diag(sqrt(eigVal))
> H <- scale(Y)                      # standardisierte Hauptkomponenten
> HB <- H %*% t(B)                   # H * B^t

# Reproduktion der Originaldaten: addiere Zentroid c: H * B^t + c
> repr <- sweep(HB, 2, ctr, "+")
> all.equal(X, repr)                  # Kontrolle: Übereinstimmung mit X
[1] TRUE
```

Sollen die ursprünglichen Daten nun im obigen Sinne optimal durch weniger Variablen repräsentiert werden, können die Spalten von \mathbf{B} von rechts kommend nacheinander gestrichen werden, wodurch sich etwa eine Matrix \mathbf{B}' ergibt. Ebenso sind die zugehörigen standardisierten Hauptkomponenten, also die Spalten von \mathbf{H} , zu streichen, wodurch die Matrix \mathbf{H}' entsteht. Die approximierten Daten $\mathbf{X}' = \mathbf{H}'\mathbf{B}'^t + \bar{x}$ liegen dann in einem affinen Unterraum mit niedrigerer Dimension als die ursprünglichen Daten (Abb. 12.2).

²⁹Zunächst gilt $\mathbf{H} = \mathbf{YD}^{-\frac{1}{2}}$, wobei $\mathbf{Y} = \dot{\mathbf{X}}\mathbf{G} = \mathbf{ZXG}$ die Matrix der (zentrierten) Hauptkomponenten und \mathbf{Z} die Zentriermatrix zu \mathbf{X} ist (vgl. Abschn. 12.1.1). Wegen $\mathbf{G}^t = \mathbf{G}^{-1}$ folgt $\mathbf{HB}^t + \bar{x} = \mathbf{YD}^{-\frac{1}{2}}(\mathbf{GD}^{\frac{1}{2}})^t + \bar{x} = \mathbf{ZXGD}^{-\frac{1}{2}}\mathbf{D}^{\frac{1}{2}}\mathbf{G}^t + \bar{x} = \dot{\mathbf{X}}\mathbf{GG}^t + \bar{x} = \mathbf{X}$.

```
# approximiere Originaldaten durch niedrig-dimensionale Daten
> HB1 <- H[ , 1] %*% t(B[ , 1]) # H' * B'^t
> repr1 <- sweep(HB1, 2, ctr, "+") # H' * B'^t + c
> sum((X-repr1)^2) # Summe der quadrierten Abweichungen
[1] 79.08294

# approximierte Daten liegen in eindimensionalem Unterraum
> qr(scale(repr1, center=TRUE, scale=FALSE))$rank
[1] 1

# grafische Darstellung der Originaldaten und der Approximation
> plot(X, xlab="x", ylab="y", pch=20, asp=1,
+       main="Originaldaten und Approximation")

> abline(coef=ab1, lwd=2, col="gray") # Achse 1
> abline(coef=ab2, lwd=2, col="gray") # Achse 2
> segments(X[ , 1], X[ , 2], repr1[ , 1], repr1[ , 2])
> points(repr1, pch=1, lwd=2, col="blue", cex=2) # Approximation
> points(ctr[1], ctr[2], pch=16, col="red", cex=3) # Zentroid

# Legende einfügen
> legend(x="topleft", legend=c("Daten", "Achsen HK",
+      "Zentroid", "Approximation"), pch=c(20, NA, 16, 1),
+      lty=c(NA, 1, NA, NA), col=c("black", "gray", "red", "blue"))
```

Die Kovarianzmatrix der Daten lässt sich als $S = BB^t$ zerlegen³⁰ (vgl. Abschn. 12.1.6). Die Reproduktion von S durch die um rechte Spalten gestrichene Matrix B' mit $B'B'^t$ wird schlechter, bleibt aber optimal im Vergleich zu allen anderen Matrizen gleicher Dimensionierung.

```
> B %*% t(B) # reproduziert Kovarianzmatrix von X
[1,] [2,]
[1,] 3.939795 2.155180
[2,] 2.155180 3.610964

> cov(X) # Kontrolle ...

# approximiere Kovarianzmatrix von X mit B' * B'^t
> B[ , 1] %*% t(B[ , 1])
[1,] [2,]
[1,] 3.194211 2.959811
[2,] 2.959811 2.742612
```

Die von `princomp()` berechnete Hauptkomponentenanalyse unterscheidet sich von der durch `prcomp()` erzeugten in den ausgegebenen Streuungen: bei `prcomp()` sind dies die korrigierten, bei `princomp()` die unkorrigierten. Die Streuungen sind gleich den zugehörigen Eigenwerten

³⁰ $BB^t = GD^{\frac{1}{2}}(GD^{\frac{1}{2}})^t = GD^{\frac{1}{2}}D^{\frac{1}{2}}G^t = GDG^t = S$.

der unkorrigierten Kovarianzmatrix der Daten.³¹ In der Ausgabe von `princomp()` sind die Koeffizienten der standardisierten Linearkombination zunächst nicht mit aufgeführt, befinden sich jedoch in der Komponente `loadings` der zurückgelieferten Liste enthalten.

```
> (pcaPrin <- princomp(X))
Call:
princomp(x = X)
Standard deviations:
  Comp.1    Comp.2
2.412071 1.257640
2 variables and 50 observations.

# Kontrolle: unkorrigierte Streuungen der Hauptkomponenten
> Sml <- cov.wt(pc, method="ML")$cov
> sqrt(diag(Sml))
      PC1        PC2
2.412071 1.257640

> pcaPrin$loadings # Koeffizientenmatrix ...
```

Die Varianzen der Hauptkomponenten werden häufig in einem Liniendiagramm (vgl. Abschn. 14.2) als Scree-Plot dargestellt, der mit `plot(<PCA>, type="b")` aufzurufen ist. Dabei ist `<PCA>` das Ergebnis einer Hauptkomponentenanalyse mit `prcomp()` oder `princomp()`. Für eine grafische Veranschaulichung der Hauptkomponenten selbst über die Hauptachsen des die gemeinsame Verteilung der Daten charakterisierenden Streuungsellipsoids vgl. Abschn. 14.6.8, Abb. 14.24. `biplot(<PCA>)` stellt die Hauptkomponenten gleichzeitig mit den Objekten in einem Diagramm dar, das eine inhaltliche Interpretation der Hauptkomponenten fördern soll.

12.3 Faktorenanalyse

Der Faktorenanalyse liegt die Vorstellung zugrunde, dass sich die korrelativen Zusammenhänge zwischen vielen beobachtbaren Merkmalen X_j (mit $j = 1, \dots, p$) aus wenigen latenten Variablen F_k (den *Faktoren* mit $k = 1, \dots, q$) speisen, die ursächlich auf die Ausprägung der X_j wirken.³² Die Wirkungsweise wird dabei als linear angenommen, d. h. die X_j sollen sich als Linearkombination der F_k zzgl. eines zufälligen Fehlers ergeben.³³ Weiterhin beruht die Faktorenanalyse auf der Annahme, dass sich bei unterschiedlichen Beobachtungsobjekten zwar die Ausprägungen der

³¹ Weiterhin basiert `prcomp()` intern auf der Singulärwertzerlegung mit `svd()`, `princomp()` hingegen auf der Berechnung der Eigenwerte mit `eigen()` (vgl. Abschn. 12.1.6). Die Singulärwertzerlegung gilt als numerisch stabiler bei schlecht konditionierten Matrizen i. S. der Kondition κ (vgl. Abschn. 12.1.5) – `prcomp()` sollte also vorgezogen werden.

³² Für weitere Verfahren, die die Beziehungen latenter und beobachtbarer Variablen modellieren, vgl. den Abschnitt *Psychometric Models* der CRAN Task Views (Mair, 2014). Lineare Strukturgleichungsmodelle werden durch die Pakete `sem` (Fox, Nie & Byrnes, 2014), `OpenMx` (Boker et al., 2011) und `lavaan` (Rosseel, 2012) unterstützt.

³³ In diesem Sinne ist die Faktorenanalyse das Gegenteil der Hauptkomponentenanalyse, in der die Hauptkomponenten Linearkombinationen der beobachtbaren Variablen sind.

F_k unterscheiden, die den Einfluss der Faktoren auf die X_j charakterisierenden Koeffizienten der Linearkombinationen dagegen fest, also für alle Beobachtungsobjekte gleich sind.

Zur Vereinbarung der Terminologie sei \mathbf{x} der p -Vektor der beobachtbaren Merkmalsausprägungen einer Person, \mathbf{f} der q -Vektor ihrer Faktorausprägungen, $\boldsymbol{\epsilon}$ der p -Vektor der Fehler und $\mathbf{\Lambda}$ die $(p \times q)$ -*Ladungsmatrix* der zeilenweise zusammengestellten Koeffizienten der Linearkombinationen für jedes X_j . Das Modell geht von standardisierten X_j und F_k aus, die also Erwartungswert 0 und Varianz 1 besitzen sollen. Weiterhin sollen die Fehler untereinander und mit den Faktoren unkorreliert sein. Das Modell lässt sich damit insgesamt als $\mathbf{x} = \mathbf{\Lambda}\mathbf{f} + \boldsymbol{\epsilon}$ formulieren.³⁴ Die X_j abzüglich des Fehlervektors seien als *reduzierte* Variablen \hat{X}_j bezeichnet, für deren Werte einer Person $\hat{\mathbf{x}} = \mathbf{\Lambda}\mathbf{f}$ gilt.

Es sind nun zwei Varianten denkbar: Zum einen kann das Modell unkorrelierter Faktoren angenommen werden, deren Korrelationsmatrix \mathbf{K}_f dann gleich der $(q \times q)$ -Einheitsmatrix ist. Dieses Modell wird auch als *orthogonal* bezeichnet, da die Faktoren in einer geeigneten geometrischen Darstellung senkrecht aufeinander stehen. In diesem Fall ist $\mathbf{\Lambda}$ gleichzeitig die auch als *Faktorstruktur* bezeichnete Korrelationsmatrix von \mathbf{x} und \mathbf{f} . Zum anderen ist das Modell potentiell korrelierter Faktoren mit Einträgen von \mathbf{K}_f ungleich 0 außerhalb der Hauptdiagonale möglich. Hier bilden die Faktoren in einer geeigneten Darstellung keinen rechten Winkel, weshalb das Modell auch als *schiefwinklig* (*oblique*) bezeichnet wird. Die Faktorstruktur berechnet sich zu $\mathbf{\Lambda}\mathbf{K}_f$, zur Unterscheidung wird $\mathbf{\Lambda}$ selbst als *Faktormuster* bezeichnet.

Für die Korrelationsmatrix \mathbf{K}_x der beobachtbaren Variablen ergibt sich im Modell korrelierter Faktoren $\mathbf{K}_x = \mathbf{\Lambda}\mathbf{K}_f\mathbf{\Lambda}^t + \mathbf{D}_\epsilon$, wenn die Diagonalmatrix \mathbf{D}_ϵ die Kovarianzmatrix der Fehler ist. Im Modell unkorrelierter Faktoren vereinfacht sich die Gleichung zu $\mathbf{K}_x = \mathbf{\Lambda}\mathbf{\Lambda}^t + \mathbf{D}_\epsilon$. Analog zum Vektor der reduzierten Variablen $\hat{\mathbf{x}}$ sei $\mathbf{K}_{\hat{\mathbf{x}}} = \mathbf{\Lambda}\mathbf{K}_f\mathbf{\Lambda}^t$ bzw. $\mathbf{K}_{\hat{\mathbf{x}}} = \mathbf{\Lambda}\mathbf{\Lambda}^t$ als reduzierte Korrelationsmatrix bezeichnet. Dabei ist zu beachten, dass $\mathbf{K}_{\hat{\mathbf{x}}}$ nicht die Korrelationsmatrix von $\hat{\mathbf{x}}$ ist – die Diagonalelemente sind nicht 1, sondern ergänzen die Diagonalelemente von \mathbf{D}_ϵ (die Fehlervarianzen) zu 1. Die Diagonalelemente von $\mathbf{K}_{\hat{\mathbf{x}}}$ heißen Kommunalitäten. Sie sind gleichzeitig die Varianzen von $\mathbf{A}\mathbf{f}$ und damit ≥ 0 .

Bei der exploratorischen Faktorenanalyse besteht der Wunsch, bei vorausgesetzter Gültigkeit eines der beiden Modelle auf Basis vieler Messwerte der X_j eine Schätzung der Ladungsmatrix $\hat{\mathbf{\Lambda}}$ sowie ggf. der Korrelationsmatrix der Faktoren $\hat{\mathbf{K}}_f$ zu erhalten.³⁵ Die Anzahl der latenten Faktoren sei dabei vorgegeben. Praktisch immer lassen sich jedoch durch Rotation viele Ladungs- und ggf. Korrelationsmatrizen finden, die zum selben Resultat i. S. von \mathbf{K}_x führen (s. u.).

Die Aufgabe kann analog zur Hauptkomponentenanalyse auch so verstanden werden, dass es die empirisch gegebene korrelative Struktur der X_j i. S. ihrer Korrelationsmatrix \mathbf{K}_x möglichst gut durch die Matrix $\hat{\mathbf{\Lambda}}$ und ggf. $\hat{\mathbf{K}}_f$ zu reproduzieren gilt: $\hat{\mathbf{\Lambda}}\hat{\mathbf{\Lambda}}^t$ bzw. $\hat{\mathbf{\Lambda}}\hat{\mathbf{K}}_f\hat{\mathbf{\Lambda}}^t$ sollte also höchstens auf der Diagonale nennenswert von \mathbf{K}_x abweichen und dort positive Einträge ≤ 1 besitzen. Die mit $\hat{\mathbf{\Lambda}}$ geschätzten Einflüsse der Faktoren auf die beobachtbaren Variablen sollen gleichzeitig dazu dienen, die Faktoren inhaltlich mit Bedeutung zu versehen.

³⁴Für n Personen seien die Werte auf den beobachtbaren Variablen zeilenweise in einer $(n \times p)$ -Matrix \mathbf{X} zusammengefasst, analog die Faktorwerte in einer $(n \times q)$ -Matrix \mathbf{F} und die Fehler in einer $(n \times p)$ -Matrix \mathbf{E} . Dann lautet das Modell $\mathbf{X} = \mathbf{F}\mathbf{\Lambda}^t + \mathbf{E}$.

³⁵Die konfirmatorische Faktorenanalyse, bei der theoretische Erwägungen ein bestimmtes, auf Konsistenz mit den Daten zu testendes $\mathbf{\Lambda}$ vorgeben, ist mit Hilfe linearer Strukturgleichungsmodelle durchzuführen (vgl. Fußnote 32).

Die Faktorenanalyse wird mit `factanal()` durchgeführt, weitere Varianten enthält das Paket `psych`.

```
> factanal(x=<Datenmatrix>, covmat=<Kovarianzmatrix>,
+           n.obs=<Anzahl Beobachtungen>, factors=<Anzahl Faktoren>,
+           scores=<Schätzung Faktorwerte>, rotation=<Rotationsart>)
```

Für `x` ist die spaltenweise aus den Daten der beobachtbaren Variablen zusammengestellte Matrix zu übergeben. Alternativ lässt sich das Argument `covmat` nutzen, um statt der Daten ihre Kovarianzmatrix zu übergeben. In diesem Fall ist für `n.obs` die Anzahl der Beobachtungen zu nennen, die `covmat` zugrunde liegen. Die gewünschte Anzahl an Faktoren muss für `factors` genannt werden. Sollen die Schätzungen der Faktorwerte \hat{f} ebenfalls berechnet werden, ist `scores` auf "regression" oder "Bartlett" zu setzen. Mit der Voreinstellung "varimax" für das Argument `rotation` liegt der Rechnung das Modell unkorrelierter Faktoren zugrunde.³⁶

Das folgende Beispiel behandelt den Fall, dass unkorrelierte Faktoren angenommen werden.

```
> N <- 200                      # Anzahl Beobachtungsobjekte
> P <- 6                        # Anzahl beobachteter Variablen
> Q <- 2                        # simulierte Anzahl von Faktoren

# hypothetische Ladungsmatrix für die Simulation der Daten
> (Lambda <- matrix(c(0.7,-0.4, 0.8,0, -0.2,0.9, -0.3,0.4,
+                     0.3,0.7, -0.8,0.1), nrow=P, ncol=Q, byrow=TRUE))
     [,1]   [,2]
[1,]  0.7  -0.4
[2,]  0.8   0.0
[3,] -0.2   0.9
[4,] -0.3   0.4
[5,]  0.3   0.7
[6,] -0.8   0.1

# hypothetische Kovarianzmatrix der Faktoren. Hier: Einheitsmatrix
> Kf <- diag(Q)                # Modell unkorrelierter Faktoren
> mu <- c(5, 15)               # hypothetisches Zentroid der Faktoren
> library(mvtnorm)             # für rmvnorm()
> FF <- rmvnorm(N, mean=mu, sigma=Kf)      # simulierte Faktorwerte
> E  <- rmvnorm(N, rep(0, P), diag(P))       # simulierte Fehler

# simulierte beobachtete Variablen im Modell der Faktorenanalyse
> X  <- FF %*% t(Lambda) + E            # Datenmatrix
> (fa <- factanal(X, factors=2, scores="regression"))
Call:
factanal(x = X, factors = 2)
```

³⁶Weitere Rotationsarten, etwa für das Modell korrelierter Faktoren, stellt das Paket `GPArotation` (Bernaards & Jennrich, 2005) zur Verfügung. Es enthält Funktionen, deren Namen an das Argument `rotation` übergeben werden können, z.B. "oblimin" für eine schiefwinklige Rotation. Für eine vollständige Liste vgl. `?rotations`, nachdem das Paket installiert und geladen wurde.

Uniquenesses:

```
[1] 0.492 0.701 0.595 0.589 0.582 0.542
```

Loadings:

	Factor1	Factor2
[1,]	0.564	-0.437
[2,]	0.546	
[3,]	-0.145	0.619
[4,]	-0.334	0.548
[5,]	0.325	0.559
[6,]	-0.674	

	Factor1	Factor2
SS loadings	1.308	1.191
Proportion Var	0.218	0.199
Cumulative Var	0.218	0.417

Test of the hypothesis that 2 factors are sufficient.

The chi square statistic is 4.74 on 4 degrees of freedom.

The p-value is 0.315

Die Ausgabe von `factanal()` liefert unter der Überschrift **Uniqueness** die geschätzten Fehlervarianzen der X_j , also die Einträge von \hat{D}_ϵ , die sich mit den Kommunalitäten zu 1 ergänzen. Die geschätzte Ladungsmatrix $\hat{\Lambda}$ ist unter **Loadings** aufgeführt, wobei nur Werte über einer bestimmten absoluten Größe angezeigt werden (Abb. 12.3). Die Faktoren sind dabei entsprechend der durch sie aufgeklärten Varianz geordnet. Die berechnete Ladungsmatrix weicht leicht von der zur Simulation verwendeten ab. Ursache hierfür ist zum einen die bereits angesprochene Uneindeutigkeit der Lösung, zum anderen der in die simulierten Daten eingeflossene Fehlerterm.

In der abschließenden Tabelle gibt **SS loadings** die jeweilige Spaltensumme der quadrierten Ladungen eines Faktors an, also die durch ihn bei allen Variablen aufgeklärte Varianz. Als Gesamtvarianz wird hier die Spur der Kovarianzmatrix der Variablen verstanden – da die Variablen standardisiert sind, ist dies gleichzeitig die Spur ihrer Korrelationsmatrix, also die Anzahl der Variablen.³⁷ **Proportion Var** ist der vom Faktor aufgeklärte Anteil an der Gesamtvarianz und **Cumulative Var** der kumulative Anteil der durch die Faktoren aufgeklärten Varianz. Die Komponente **scores** der von `factanal()` erzeugten Liste enthält die hier über die Regressionsmethode geschätzten Faktorwerte. Die von `factanal()` ggf. verwendete Rotationsmatrix findet sich in der Komponente **rotmat**.

```
# Spaltensumme der quadrierten Ladungen -> aufgeklärte Varianz
> Lhat <- fa$loadings                      # geschätzte Ladungsmatrix
> colSums(Lhat^2)
```

³⁷ Setzt man `rotation="none"`, ist dies ist bei der durch `factanal()` verwendeten Methode, um ein $\hat{\Lambda}$ zu erzeugen, gleichzeitig der zugehörige Eigenwert der geschätzten reduzierten Korrelationsmatrix $\hat{K}_{\hat{x}} = \hat{\Lambda}\hat{\Lambda}^t$. Bei der Methode handelt es sich um die iterative Maximum-Likelihood Kommunalitätenschätzung. Bei Rotation oder anderen Schätzmethoden gilt diese Gleichheit dagegen nicht.

```

Factor1    Factor2
1.307778  1.191397

# Spaltensummen der quadrierten Ladungen geteilt durch Spur der
# Korrelationsmatrix -> Anteil der aufgeklärten Varianz
> colSums(Lhat^2) / sum(diag(cor(X)))
  Factor1    Factor2
0.2179629  0.1985661

> head(fa$scores)                      # geschätzte Faktorwerte
  Factor1    Factor2
[1,] -1.1439721  0.4063908
[2,] -1.2996309 -0.4458015
[3,]  0.9340950 -0.4548785
[4,] -0.8481367 -0.3690898
[5,] -0.3358804 -0.2208062
[6,]  0.9145849 -1.0632808

```

Schließlich folgt in der Ausgabe von `factanal()` die χ^2 -Teststatistik für den Test der H_0 , dass das Modell mit der angegebenen Zahl an Faktoren tatsächlich gilt. Ein kleiner p -Wert deutet daraufhin, dass die Daten nicht mit einer perfekten Passung konsistent sind. Weitere Hilfestellung zur Frage, wieviele Faktoren zu wählen sind, liefert etwa das Kaiser-Guttman-Kriterium, dem zufolge nur so viele Faktoren zu berücksichtigen sind, wie es Eigenwerte von \mathbf{K}_x größer 1 gibt, dem Mittelwert dieser Eigenwerte (Abb. 12.3).³⁸ Eine andere Herangehensweise besteht darin, den abfallenden Verlauf der Eigenwerte von \mathbf{K}_x zu betrachten, um einen besonders markanten Sprung zu identifizieren (Scree-Test). Hierbei hilft die grafische Darstellung dieser Eigenwerte als Liniendiagramm im Scree-Plot (Abb. 12.3).³⁹

Mit dem von `factanal()` zurückgegebenen Objekt sollen nun die Kommunalitäten sowie die geschätzte Korrelationsmatrix der beobachtbaren Variablen ermittelt werden. Schließlich folgt die grafische Darstellung der Faktorladungen der Variablen (Abb. 12.3).

```

> 1 - fa$uniqueness                  # Kommunalitäten: 1 - Fehlervarianzen
[1] 0.5084807 0.2985865 0.4045587 0.4113273 0.4180855 0.4581350

> rowSums(Lhat^2)                    # Zeilensummen der quadrierten Ladungen
[1] 0.5084808 0.2985873 0.4045598 0.4113266 0.4180855 0.4581344

# geschätzte Korrelationsmatrix der Variablen: Lambda * Lambda^t + D_e
> KxEst <- Lhat %*% t(Lhat) + diag(fa$uniqueness)

# grafische Darstellung der Faktorladungen

```

³⁸Die Summe der Eigenwerte einer Matrix ist gleich deren Spur (vgl. Abschn. 12.1.5) – im Fall der Korrelationsmatrix \mathbf{K}_x also gleich der Anzahl der Variablen p , da in der Diagonale überall 1 steht. Der Mittelwert der Eigenwerte ist damit $\frac{1}{p}p = 1$.

³⁹Weitere Verfahren, die der Klärung der geeigneten Anzahl von Faktoren dienen sollen, sind die Parallelanalyse mit `fa.parallel()` aus dem Paket `psych` sowie das *very simple structure* Verfahren mit `VSS()` aus demselben Paket.

```
> plot(Lhat, xlab="Faktor 1", ylab="Faktor 2",
+      xlim=c(-1.1, 1.1), ylim=c(-1.1, 1.1), pch=20, asp=1,
+      main="Faktorenanalyse: Ladungen und (rotierte) Faktoren")

> abline(h=0, v=0) # Achsen mit Ursprung 0

# Faktoren einzeichnen
> arrows(0, 0, c(1, 0), c(0, 1), col="blue", lwd=2)
> angles <- seq(0, 2*pi, length.out=200) # Winkel Einheitskreis
> circ <- cbind(cos(angles), sin(angles)) # Koordinaten Einheitskr.
> lines(circ) # Einheitskreis
```

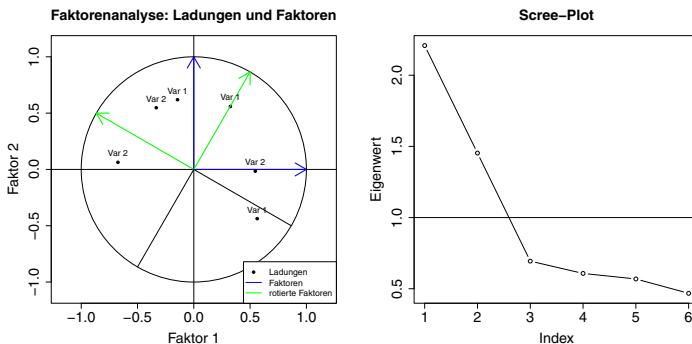


Abbildung 12.3: Faktorenanalyse: Faktorladungen der Variablen sowie rotierte Faktoren. Scree-Plot der Eigenwerte der Korrelationsmatrix der Daten

Durch orthogonale oder schiefwinklige Rotation der Faktoren lassen sich aus der von `factanal()` geschätzten Ladungsmatrix und den zugehörigen Faktorwerten weitere Ladungsmatrizen berechnen, die eine ebenso gute Reproduktion der beobachteten Korrelationsmatrix \mathbf{K}_x liefern. Ist allgemein die Rotationsmatrix \mathbf{G} eine Orthogonalmatrix ($\mathbf{G}^t = \mathbf{G}^{-1}$), ergibt sich die neue geschätzte Ladungsmatrix $\hat{\Lambda}$ im Modell unkorrelierter Faktoren aus der alten durch $\hat{\Lambda} = \hat{\Lambda}\mathbf{G}$ (Abb. 12.3). Die neue geschätzte Korrelationsmatrix $\hat{\Lambda}\hat{\Lambda}^t + \hat{D}_\epsilon$ stimmt dann mit $\hat{\Lambda}\hat{\Lambda}^t + \hat{D}_\epsilon$ überein.

```
> ang <- pi/3 # Rotationswinkel

# Matrix für orthogonale Rotation der Faktoren
> G <- matrix(c(cos(ang), sin(ang), -sin(ang), cos(ang)), nrow=2)
> (Lrot <- Lhat %*% G) # neue Ladungsmatrix
[,1] [,2]
[1,] -0.09662051 -0.706502158
[2,] 0.25977831 -0.480731207
[3,] 0.46393903 0.435109654
[4,] 0.30755796 0.562791872
[5,] 0.64659330 -0.001618208
[6,] -0.28224873 0.615199187
```

```
# neue geschätzte Korrelationsmatrix der beobachtbaren Variablen
> KxEstRot <- Lrot %*% t(Lrot) + diag(fa$uniquenesses)
> all.equal(KxEst, KxEstRot)      # Kontrolle: stimmt mit alter überein
[1] TRUE

# rotierte Faktoren einzeichnen
> arrows(0, 0, G[1, ], G[2, ], col="green", lwd=2)
> segments(0, 0, -G[1, ], -G[2, ])

# Variablen beschriften und Legende einfügen
> text(Lhat[, 1], Lhat[, 2]+0.06, labels=paste("Var", 1:Q))
> legend(x="bottomright", legend=c("Ladungen", "Faktoren",
+ "rotierte Faktoren"), pch=c(20, NA, NA),
+ lty=c(NA, 1, 1), col=c("black", "blue", "green"))

# Scree-Plot der Eigenwerte der Korrelationsmatrix der Daten
> plot(eigen(cor(X))$values, type="b", ylab="Eigenwert",
+       main="Scree-Plot")

> abline(h=1)                  # Referenzlinie für Kaiser-Kriterium
```

Ist G eine schiefwinklige Rotationsmatrix, ergibt sich die neue geschätzte Ladungsmatrix $\tilde{\Lambda}$ aus der alten durch $\tilde{\Lambda} = \hat{\Lambda}G^{t-1}$. Die neue geschätzte Korrelationsmatrix der Faktoren ist $\tilde{K}_f = G'\hat{K}_fG$. Die geschätzte Faktorstruktur, also die Matrix der Korrelationen zwischen beobachtbaren Variablen und Faktoren, berechnet sich hier als $\hat{\Lambda}\hat{K}_fG$. Die neue geschätzte Korrelationsmatrix $\tilde{\Lambda}\tilde{K}_f\tilde{\Lambda}^t + \hat{D}_\epsilon$ stimmt dann mit $\hat{\Lambda}\hat{K}_f\hat{\Lambda}^t + \hat{D}_\epsilon$ überein.

```
# Matrix für schiefwinklige Rotation der Faktoren
> G <- matrix(c(5/3, -1, -4/3, 8/5), nrow=2, byrow=TRUE)

# neue Korrelationsmatrix der Faktoren (bisher: Einheitsmatrix)
> KfRot <- t(G) %*% diag(Q) %*% G

# neue geschätzte Ladungsmatrix
> Lrot <- Lhat %*% solve(t(G))

# neue geschätzte Faktorstruktur
> facStruct <- Lhat %*% diag(Q) %*% G

# neue geschätzte Korrelationsmatrix der beobachtbaren Variablen
> KxEstRot <- Lrot %*% KfRot %*% t(Lrot) + diag(fa$uniquenesses)
> all.equal(KxEst, KxEstRot)      # Kontrolle: stimmt mit alter überein
[1] TRUE
```

12.4 Multidimensionale Skalierung

Die multidimensionale Skalierung ist ein weiteres Verfahren zur Dimensionsreduktion, das bestimmte Relationen zwischen Objekten durch deren Anordnung auf möglichst wenigen Merkmalsdimensionen zu repräsentieren sucht. Die betrachtete Eigenschaft ist hier die globale Unähnlichkeit von je zwei Objekten. Die Ausgangssituation unterscheidet sich von jener in der Hauptkomponentenanalyse und der Faktorenanalyse dahingehend, dass zunächst unbekannt ist, bzgl. welcher Merkmale Objekte beurteilt werden, wenn eine Aussage über ihre generelle Ähnlichkeit zu anderen Objekten getroffen wird. Anders formuliert sind Anzahl und Bedeutung der Variablen, für die Objekte Werte besitzen, nicht gegeben. Dementsprechend werden in einer empirischen Erhebung oft auch nicht separat bestimmte Eigenschaften von einzelnen Objekten gemessen – vielmehr gilt es, in einem Paarvergleich je zwei Objekte hinsichtlich ihrer Unähnlichkeit zu beurteilen.

Die metrische multidimensionale Skalierung sucht nach Merkmalsdimensionen, auf denen sich die Objekte als Punkte so anordnen lassen, dass die paarweisen Unähnlichkeitswerte möglichst gut mit dem euklidischen Abstand zwischen den Punkten übereinstimmen.⁴⁰ Sie wird mit `cmdscale()` durchgeführt.

```
> cmdscale(d=(Distanzmatrix), k=(Anzahl an Dimensionen), x.ret=FALSE)
```

Als Argument `d` ist eine symmetrische Matrix zu übergeben, deren Zeilen und Spalten dieselben Objekte repräsentieren. Die Abstände zwischen verschiedenen Objekten i. S. von Werten eines Unähnlichkeitsmaßes sind in den Zellen außerhalb der Hauptdiagonale enthalten, die selbst überall 0 ist. Liegen als Daten nicht bereits die Abstände zwischen Objekten vor, sondern die separat für alle Objekte erhobenen Werte bzgl. derselben Variablen, können diese mit `dist()` in eine geeignete Distanzmatrix transformiert werden (vgl. Abschn. 12.1.3). Für `k` ist anzugeben, durch wieviele Variablen der Raum aufgespannt werden soll, in dem `cmdscale()` die Objekte anordnet und ihre euklidischen Distanzen berechnet. Liegen Unähnlichkeitswerte zwischen n Objekten vor, kann die gewünschte Dimension höchstens $n - 1$ sein, Voreinstellung ist 2.

Die Ausgabe umfasst eine $(n \times k)$ -Matrix mit den n Koordinaten der Objekte im k -dimensionalen Variablenraum, wobei das Zentroid im Ursprung des Koordinatensystems liegt. Mit `x.ret=TRUE` enthält das Ergebnis zwei zusätzliche Informationen: die Matrix der paarweisen Distanzen der Objekte bzgl. der ermittelten Merkmalsdimensionen und ein Maß für die Güte der Anpassung der ursprünglichen Unähnlichkeiten durch die euklidischen Distanzen. Das Ergebnis ist insofern uneindeutig, als Rotation, Spiegelung und gleichförmige Verschiebung der Punkte ihre Distanzen zueinander nicht ändern.

Als Beispiel liegen die Straßen-Entferungen zwischen deutschen Städten vor, die die multidimensionale Skalierung auf zwei Dimensionen räumlich anordnen soll.

```
# Städte, deren Entferungen berücksichtigt werden
> cities <- c("Augsburg", "Berlin", "Dresden", "Hamburg", "Hannover",
+           "Karlsruhe", "Kiel", "München", "Rostock", "Stuttgart")

# erstelle Distanzmatrix aus Vektor der Entferungen
> n      <- length(cities)                                # Anzahl der Städte
```

⁴⁰Die nichtmetrische multidimensionale Skalierung wird durch `monoMDS()` aus dem `vegan` Paket bereit gestellt.

```

> dstMat <- matrix(numeric(n^2), nrow=n)      # zunächst leere Matrix
> cityDst <- c(596, 467, 743, 599, 226, 838, 65, 782, 160, 194, 288,
+           286, 673, 353, 585, 231, 633, 477, 367, 550, 542, 465,
+           420, 510, 157, 623, 96, 775, 187, 665, 480, 247, 632,
+           330, 512, 723, 298, 805, 80, 872, 206, 752, 777, 220, 824)

> dstMat[upper.tri(dstMat)] <- rev(cityDst)
> dstMat <- t(dstMat[, n:1])[ , n:1]
> dstMat[lower.tri(dstMat)] <- t(dstMat)[lower.tri(dstMat)]

# füge Städtenamen in Zeilen und Spalten hinzu
> dimnames(dstMat) <- list(city=cities, city=cities)
> (mds <- cmdscale(dstMat, k=2))      # multidimensionale Skalierung
   [,1]      [,2]
Augsburg  399.60802 -70.51443
Berlin    -200.20462 -183.39465
Dresden   -18.47337 -213.01950
Hamburg   -316.39991 130.72245
Hannover  -161.38209 120.21761
Karlsruhe 333.38724 212.12523
Kiel      -409.08703 147.62226
München   408.55752 -144.46446
Rostock   -401.19605 -126.20651
Stuttgart 365.19030 126.91200

```

Eine grafische Darstellung der ermittelten Koordinaten erlaubt es, das Ergebnis mit den tatsächlichen Positionen zu vergleichen (Abb. 12.4). Hier ist zu erkennen, dass das Ergebnis für die meisten Städte gut mit der realen Topografie übereinstimmt, wenn man die Möglichkeit einer Drehung im Uhrzeigersinn um 90° berücksichtigt und dann die West-Ost-Richtung spiegelt.

```

> xLims <- range(mds[, 1]) + c(0, 250)      # x-Achsenbereich
> plot(mds, xlim=xLims, xlab="Nord-Süd", ylab="Ost-West", pch=16,
+       main="Anordnung der Städte nach MDS")

# füge Städtenamen hinzu
> text(mds[, 1]+50, mds[, 2], adj=0, labels=cities)

```

12.5 Multivariate multiple Regression

Die univariate multiple Regression (vgl. Abschn. 6.3) lässt sich zur multivariatenen multiplen Regression verallgemeinern, bei der durch p Prädiktoren X_j (mit $j = 1, \dots, p$) nicht nur ein Kriterium Y vorhergesagt werden soll, sondern r Kriteriumsvariablen Y_l (mit $l = 1, \dots, r$) gleichzeitig. Die Parameterschätzung im multivariaten Fall ist jedoch auf den univariaten zurückführbar: Die i.S. der geringsten Quadratsumme der Residuen optimale Parameterwahl geht aus der Zusammenstellung der r unabhängig voneinander durchgeföhrten Regressionen von jeweils einem Kriterium Y_l auf alle Prädiktoren X_j hervor.

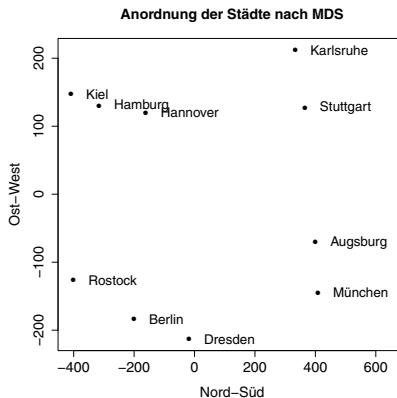


Abbildung 12.4: Ergebnis der multidimensionalen Skalierung auf Basis der Distanzen zwischen deutschen Städten

In der Berechnung der multivariaten multiplen Regression mittels `lm()` ist die Modellformel zunächst wie im univariaten Fall aufzubauen. Im Unterschied dazu ist das Kriterium auf der linken Seite der Formel hier jedoch kein Vektor, sondern muss eine spaltenweise aus den einzelnen Kriteriumsvariablen zusammengestellte Matrix sein.

Im Beispiel sollen anhand der Prädiktoren Alter, Körpergröße und wöchentliche Dauer sportlicher Aktivitäten die Kriterien Körpergewicht und Gesundheit (i. S. eines geeigneten quantitativen Maßes) vorhergesagt werden.

```
> N      <- 100                      # Anzahl Versuchspersonen
> height <- rnorm(N, 175, 7)        # Prädiktor 1
> age    <- rnorm(N, 30, 8)         # Prädiktor 2
> sport   <- abs(rnorm(N, 60, 30))  # Prädiktor 3

# modellgerechte Simulation beider Kriterien
> weight <- 0.5*height - 0.3*age - 0.4*sport + 10 + rnorm(N, 0, 3)
> health <- -0.3*age + 0.6*sport + rnorm(N, 4)
> Y      <- cbind(weight, health)    # Matrix der Kriterien

# multivariat formulierte Modellformel für lm()
> (fitM <- lm(Y ~ height + age + sport))
Call:
lm(formula = Y ~ height + age + sport)

Coefficients:
```

	weight	health
(Intercept)	10.41235	2.44980
height	0.50721	0.01031
age	-0.34730	-0.29678

```

sport      -0.40894  0.59826

# Kontrolle: Koeffizienten beider univariater Regressionen separat
> coef(lm(weight ~ height + age + sport))
(Intercept) height      age      sport
10.4124    0.5072   -0.3473  -0.4089

> coef(lm(health ~ height + age + sport))
Coefficients:
(Intercept) height      age      sport
2.44980    0.01031   -0.29678  0.59826

```

Die multivariate Regressionsanalyse als inferenzstatistischer Test liefert i. d. R. andere Ergebnissen als die separaten univariaten Regressionsanalysen (vgl. Abschn. 12.9.6, 12.9.7). Das von `lm()` erzeugte Objekt muss dazu an die Funktion `summary(manova())` übergeben werden, die `aov()` (vgl. Abschn. 7.3.2) auf den multivariaten Fall verallgemeinert und ebenso wie diese aufzurufen ist. Mit dem Argument `test` von `summary()` lassen sich verschiedene multivariate Teststatistiken wählen, etwa die Hotelling-Lawley-Spur mit "Hotelling-Lawley" (für weitere vgl. `?summary.manova` und Abschn. 12.9.7). Anders als bei den univariaten Tests ist beim multivariaten Test der Parameter die Reihenfolge der Prädiktoren relevant (für die manuelle Kontrolle vgl. Abschn. 12.9.8).

```

> summary(manova(fitM), test="Hotelling-Lawley")
      Df Hotelling-Lawley approx F num Df den Df     Pr(>F)
height     1             8.056    382.7      2     95 < 2.2e-16 ***
age        1             6.897    327.6      2     95 < 2.2e-16 ***
sport      1            257.924   12251.4     2     95 < 2.2e-16 ***
Residuals 96

> summary(manova(fitM), test="Wilks")           # Wilks' Lambda ...
> summary(manova(fitM), test="Roy")             # Roys Maximalwurzel ...
> summary(manova(fitM), test="Pillai")          # Pillai-Bartlett-Spur ...

```

12.6 Hotellings T^2

12.6.1 Test für eine Stichprobe

Hotellings T^2 -Test für eine Stichprobe prüft die Datenvektoren mehrerer gemeinsam normalverteilter Variablen daraufhin, ob sie mit der H_0 konsistent sind, dass ihr theoretisches Zentroid mit einem bestimmten Vektor übereinstimmt. Der Test lässt sich mit der Funktion `HotellingsT2Test()` aus dem Paket `DescTools` durchführen. Sie arbeitet analog zu `t.test()` für den univariaten t -Test (vgl. Abschn. 7.2). Der zweiseitige t -Test ist zu Hotellings T^2 -Test äquivalent, wenn nur Daten einer AV vorliegen.

```
> HotellingsT2Test(X=(Datenmatrix), mu=(Zentroid unter H0))
```

Unter X ist die Datenmatrix einzutragen, bei der sich die Beobachtungsobjekte in den Zeilen und die Variablen in den Spalten befinden. Das Argument mu legt das theoretische Zentroid unter H_0 fest.

Im Beispiel sollen die Werte zweier Variablen betrachtet werden. Zunächst sind nur die Daten aus einer Stichprobe (von später dreien) relevant.

```
# theoretische 2x2 Kovarianzmatrix für Zufallsvektoren
> sigma <- matrix(c(16,-2, -2,9), byrow=TRUE, ncol=2)
> mu11  <- c(-4, 4)                                # theoretisches Zentroid
> Nj    <- c(15, 25, 20)                            # Gruppengrößen

# Datenmatrix für 1. Gruppe mit Variablen in den Spalten
> library(mvtnorm)                                 # für rmvnorm()
> Y11  <- round(rmvnorm(Nj[1], mean=mu11, sigma=sigma))
> muH0 <- c(-3, 2)                                # Zentroid unter H0
> library(DescTools)                               # für HotellingsT2Test()
> HotellingsT2Test(Y11, mu=muH0)
Hotelling's one sample T2-test
data: Y11
T.2 = 9.0917, df1 = 2, df2 = 13, p-value = 0.003390
alternative hypothesis: true location is not equal to c(-3,2)
```

Die Ausgabe nennt den empirischen Wert der Teststatistik (T.2), bei dem es sich nicht um Hotellings T^2 selbst, sondern bereits um die transformierten Statistik handelt, die dann F -verteilt ist. Weiterhin sind die Freiheitsgrade der zugehörigen F -Verteilung (df1, df2) und der entsprechende p -Wert (p-value) aufgeführt.

Alternativ lässt sich der Test auch mit der aus der univariaten Varianzanalyse bekannten Funktion `anova(lm(<Modellformel>))` durchführen (vgl. Abschn. 7.3.3). In der für `lm()` anzugebenden Fomel $\langle AV \rangle \sim \langle UV \rangle$ sind die $\langle AV \rangle$ Werte dabei multivariat, d. h. in Form einer spaltenweise aus den einzelnen Variablen zusammengestellten Matrix zu übergeben. Von ihren Zeilenvektoren muss das Zentroid unter H_0 abgezogen werden. Der rechte $\langle UV \rangle$ Teil der Modellformel besteht hier nur aus dem absoluten Term 1. Als weitere Änderung lässt sich im multivariaten Fall das Argument test von `anova()` verwenden, um eine von mehreren multivariaten Teststatistiken auszuwählen.⁴¹ Dies kann "Hotelling-Lawley" für die Hotelling-Lawley-Spur sein, wobei die Wahl hier nicht relevant ist: Wenn nur eine Bedingung vorliegt, sind alle auswählbaren Teststatistiken zu Hotellings T^2 -Statistik äquivalent.

```
# ziehe Zentroid unter H0 von allen Zeilenvektoren ab
> Y11ctr <- scale(Y11, center=muH0, scale=FALSE)
> (anRes <- anova(lm(Y11ctr ~ 1), test="Hotelling-Lawley"))
Df Hotelling-Lawley approx F num Df den Df   Pr(>F)
(Intercept) 1          1.3987   9.0917      2       13 0.003390 ***
Residuals   14
```

⁴¹Bei der multivariaten Formulierung des Modells wird intern aufgrund der generischen `anova()` Funktion automatisch `anova.mlm()` verwendet, ohne dass dies explizit angegeben werden muss (vgl. Abschn. 15.2.6).

Die Ausgabe nennt den Wert der F -verteilten transformierten T^2 -Teststatistik in der Spalte `approx F`. Das Ergebnis lässt sich auch manuell prüfen. Dabei kann verifiziert werden, dass der nicht transformierte T^2 -Wert gleich dem n -fachen der quadrierten Mahalanobisdistanz zwischen dem Zentroid der Daten und dem Zentroid unter H_0 bzgl. der korrigierten Kovarianzmatrix der Daten ist. Außerdem ist T^2 gleich dem $(n - 1)$ -fachen der Hotelling-Lawley-Spur.

```
> n      <- nrow(Y11)                      # Stichprobengröße
> ctr   <- colMeans(Y11)                   # Zentroid

# Hotellings T^2 Teststatistik
> (T2 <- n * (t(ctr-muH0) %*% solve(cov(Y11)) %*% (ctr-muH0)))
[1,] 19.58203

# Kontrolle: n-faches der quadrierten Mahalanobisdistanz
> n * mahalanobis(ctr, muH0, cov(Y11))
[1] 19.58203

# Kontrolle: (n-1)-faches der Hotelling-Lawley-Spur
> (n-1) * anRes[1, "Hotelling-Lawley"]
[1] 19.58203

> P      <- ncol(Y11)                     # Anzahl Variablen
> (T2f <- ((n-P) / (P*(n-1))) * T2)       # transformiertes T^2
[1,] 9.091655

> (pVal <- pf(T2f, P, n-P, lower.tail=FALSE)) # p-Wert
[1,] 0.003389513
```

12.6.2 Test für zwei unabhängige Stichproben

Hotellings T^2 -Test für zwei unabhängige Stichproben prüft die in zwei Bedingungen erhobenen Datenvektoren mehrerer gemeinsam normalverteilter Variablen mit identischen Kovarianzmatrizen daraufhin, ob sie mit der H_0 konsistent sind, dass ihre theoretischen Zentroide übereinstimmen. Der Test ist äquivalent zum univariaten t -Test für zwei unabhängige Stichproben, wenn nur Werte einer AV vorliegen (vgl. Abschn. 7.2.2). Hotellings T^2 -Test lässt sich mit `HotellingsT2Test()` aus dem Paket `DescTools` durchführen.

```
> HotellingsT2Test(X=<Datenmatrix>, Y=<Datenmatrix>)
```

Unter `X` ist die Datenmatrix aus der ersten Bedingung einzutragen, bei der sich die Beobachtungsobjekte in den Zeilen und die Variablen in den Spalten befinden. Für `Y` ist die ebenso aufgebaute Datenmatrix aus der zweiten Bedingung zu nennen.

Das im vorangehenden Abschnitt begonnene Beispiel soll nun um die in einer zweiten Bedingung erhobenen Daten der betrachteten Variablen erweitert werden.

```
> mu21 <- c(3, 3)                         # Zentroid Zufallsvektoren 2. Bedingung
```

```
# Datenmatrix aus 2. Bedingung mit Variablen in den Spalten
> library(mvtnorm)                      # für rmvnorm()
> Y21 <- round(rmvnorm(Nj[2], mean=mu21, sigma=sigma))
> library(DescTools)                     # für HotellingsT2Test()
> HotellingsT2Test(Y11, Y21)
Hotelling's two sample T2-test
data: Y11 and Y21
T.2 = 11.235, df1 = 2, df2 = 37, p-value = 0.0001539
alternative hypothesis: true location difference is not equal to c(0,0)
```

Die Ausgabe nennt den empirischen Wert der bereits transformierten Teststatistik ($T.2$, s. o.), die Freiheitsgrade der passenden F -Verteilung ($df1, df2$) und den zugehörigen p -Wert ($p\text{-value}$).

Alternativ lässt sich der Test auch mit `anova(lm(<Modellformel>))` durchführen. Dabei sind in der Modellformel $\langle AV \rangle \sim \langle UV \rangle$ die $\langle AV \rangle$ Werte beider Gruppen in Form einer spaltenweise aus den einzelnen Variablen zusammengestellten Matrix zu übergeben, deren Zeilen von den Beobachtungsobjekten gebildet werden. $\langle UV \rangle$ codiert in Form eines Faktors für jede Zeile der $\langle AV \rangle$ Matrix, aus welcher Bedingung der zugehörige Datenvektor stammt. Das Argument `test` von `anova()` kann auf "Hotelling-Lawley" für die Hotelling-Lawley-Spur gesetzt werden, wobei auch bei zwei Gruppen alle multivariaten Teststatistiken äquivalent sind.

```
> Yht <- rbind(Y11, Y21)                 # Gesamt-Datenmatrix beider Bedingungen

# codiere für jede Zeile der Datenmatrix die zugehörige Bedingung
> IVht <- factor(rep(1:2, Nj[1:2]))
> anova(lm(Yht ~ IVht), test="Hotelling-Lawley")
Analysis of Variance Table
      Df Hotelling-Lawley approx F num Df den Df   Pr(>F)
(Intercept) 1            2.4627    45.560     2      37 1.049e-10 ***
IVht         1            0.6073    11.235     2      37 0.0001539 ***
Residuals   38
```

Die Ausgabe gleicht jener der unvarianten Anwendung von `anova()`, die Kennwerte des Tests des Faktors stehen in der mit seinem Namen bezeichneten Zeile.

Schließlich existiert mit `manova()` eine Funktion, die `aov()` (vgl. Abschn. 7.3.2) auf den multivariaten Fall verallgemeinert und ebenso wie diese aufzurufen ist. Die Modellformel ist multivariat wie mit `lm(<Modellformel>)` zu formulieren (vgl. Abschn. 12.5). In der Anwendung von `summary()` auf das von `manova()` erzeugte Modell wird hier wieder als Teststatistik die Hotelling-Lawley-Spur gewählt.

```
> (sumRes <- summary(manova(Yht ~ IVht), test="Hotelling-Lawley"))
      Df Hotelling-Lawley approx F num Df den Df   Pr(>F)
IVht         1            0.6073    11.235     2      37 0.0001539 ***
Residuals  38
```

Das Ergebnis lässt sich auch manuell prüfen. Dabei kann verifiziert werden, dass der nicht transformierte T^2 -Wert bis auf einen Faktor gleich der quadrierten Mahalanobisdistanz beider

Zentroide bzgl. einer geeigneten Schätzung der Kovarianzmatrix der Differenzvektoren ist. Außerdem ist T^2 gleich dem $(n_1 + n_2 - 2)$ -fachen der Hotelling-Lawley-Spur.

```
> n1 <- nrow(Y11)                                # Gruppengröße 1
> n2 <- nrow(Y21)                                # Gruppengröße 2
> ctr1 <- colMeans(Y11)                            # Zentroid 1. Bedingung
> ctr2 <- colMeans(Y21)                            # Zentroid 2. Bedingung

# unkorrigierte Kovarianzmatrizen aus beiden Bedingungen
> S1 <- cov.wt(Y11, method="ML")$cov
> S2 <- cov.wt(Y21, method="ML")$cov

# mit Stichprobengröße gewichtete Summe der Kovarianzmatrizen
> Su <- (1 / (n1+n2-2)) * (n1*S1 + n2*S2)

# Hotellings T^2
> (T2 <- ((n1*n2) / (n1+n2)) * (t(ctrl2-ctrl1) %*% solve(Su) %*% (ctrl2-ctrl1)))
[1,] 23.07731

# Kontrolle: quadrierte Mahalanobisdistanz beider Zentroide bzgl. Su
> ((n1*n2) / (n1+n2)) * mahalanobis(ctrl1, ctrl2, Su)
[1] 23.07731

# Kontrolle: T^2 = (n1+n2-2) * Hotelling-Lawley-Spur
> (n1+n2-2) * sumRes$stats["IVht", "Hotelling-Lawley"]
[1] 23.07731

> P      <- ncol(Y11)                            # Anzahl Variablen
> (T2f <- ((n1+n2-P-1) / ((n1+n2-2)*P)) * T2)    # transformiertes T^2
[1,] 11.23501

> (pVal <- pf(T2f, P, n1+n2-P-1, lower.tail=FALSE))      # p-Wert
[1,] 0.0001538927
```

12.6.3 Test für zwei abhängige Stichproben

Analog zum univariaten t -Test (vgl. Abschn. 7.2.3) kann der multivariate T^2 -Test für zwei abhängige Stichproben auf die Situation einer Stichprobe zurückgeführt werden: Dazu bildet man variablenweise pro Beobachtungsobjekt die Differenz der Daten aus beiden Bedingungen und stellt die Differenzvariablen spaltenweise zu einer neuen Matrix zusammen. Hotellings T^2 -Test für eine Stichprobe ist mit diesen Differenzdaten dann mit der H_0 durchzuführen, dass ihr theoretisches Zentroid der Vektor $\mathbf{0}$ ist.

```
> N      <- 20                                     # Anzahl Personen
> P      <- 2                                      # Anzahl Messzeitpunkte
> Y1t0 <- rnorm(N, mean=90,  sd=15)                # AV 1 Zeitpunkt t0
> Y1t1 <- rnorm(N, mean=100, sd=15)                 # AV 1 Zeitpunkt t1
```

```

> Y2t0 <- rnorm(N, mean=85, sd=15)           # AV 2 Zeitpunkt t0
> Y2t1 <- rnorm(N, mean=105, sd=15)           # AV 2 Zeitpunkt t1

# Faktor, der im Long-Format den Messzeitpunkt codiert
> IV <- factor(rep(1:P, each=N), labels=c("t0", "t1"))
> id <- factor(rep(1:N, times=P))            # Faktor Personen-ID

# Datensatz im Long-Format
> Ydf <- data.frame(id, Y1=c(Y1t0, Y1t1), Y2=c(Y2t0, Y2t1), IV)

# bilde pro Variable personenweise Differenz zwischen t0 und t1
> dfDiff <- aggregate(cbind(Y1, Y2) ~ id, data=Ydf, FUN=diff)
> DVdiff <- data.matrix(dfDiff[ , -1])          # als Matrix
> muH0   <- c(0, 0)                            # Zentroid unter H0
> library(DescTools)                           # für HotellingsT2Test()
> HotellingsT2Test(DVdiff, mu=muH0)
Hotelling's one sample T2-test
data: DVdiff
T.2 = 7.8777, df1 = 2, df2 = 18, p-value = 0.003486
alternative hypothesis: true location is not equal to c(0,0)

```

Auch hier kann alternativ die multivariate Variante der `anova()` Funktion verwendet werden, deren mögliche Teststatistiken im Fall einer Gruppe alle zu T^2 äquivalent sind. Da das theoretische Zentroid unter H_0 gleich **0** ist, entfällt hier die Notwendigkeit, es zuvor von den Zeilenvektoren der Datenmatrix der Differenzvariablen abzuziehen.

```

> anova(lm(DVdiff ~ 1), test="Hotelling-Lawley")
Analysis of Variance Table
      Df Hotelling-Lawley approx F num Df den Df Pr(>F)
(Intercept) 1          0.87529    7.8777      2     18 0.003486 ***
Residuals   19

```

12.6.4 Univariate Varianzanalyse mit abhängigen Gruppen (RB- p)

Daten einer eigentlich univariaten Varianzanalyse mit p abhängigen Gruppen (RB- p Design, vgl. Abschn. 7.4) können auch multivariat ausgewertet werden, wodurch die Voraussetzung der Zirkularität entfällt. Hierfür sind zunächst blockweise alle $\binom{p}{2} = \frac{p(p-1)}{2}$ Differenzvariablen zwischen je zwei Gruppen zu bilden und spaltenweise zu einer Matrix zusammenzufassen. Deren Spalten sind für $p > 2$ jedoch linear abhängig, da es höchstens $p - 1$ linear unabhängige Differenzvariablen gibt. Um diese Redundanz zu beseitigen, müssen $p - 1$ Spalten der Matrix ausgewählt werden, in deren Differenzen insgesamt alle p Gruppen eingeflossen sind. Die übrigen Spalten der ursprünglichen Matrix werden gestrichen. Für die beibehaltenen Differenzvariablen ist Hotellings T^2 -Test für eine Stichprobe mit der H_0 durchzuführen, dass ihr Zentroid der Vektor **0** ist. Das Vorgehen ist damit analog zu jenem bei Hotellings T^2 -Test für zwei abhängige Stichproben.

Als Beispiel diene jenes aus Abschn. 7.4 mit einer zu vier Messzeitpunkten erhobenen AV.

```

> P      <- 4                                # Anzahl Messzeitpunkte
> DVw   <- cbind(DV_t1, DV_t2, DV_t3, DV_t4)    # Datenmatrix Wide-Format

# alle paarweisen Differenzen der Spalten der Datenmatrix
> diffMat <- combin(1:P, 2, function(x) { DVw[ , x[1]] - DVw[ , x[2]] } )

# wähle p-1 Differenzvariablen, die nicht redundant sein dürfen
> DVdiff <- diffMat[ , 1:(P-1)]                # Zentroid unter H0
> muH0   <- rep(0, ncol(DVdiff))                 # für HotellingsT2Test()
> library(DescTools)
> HotellingsT2Test(DVdiff, mu=muH0)
Hotelling's one sample T2-test
data: DVdiff
T.2 = 2.9106, df1 = 3, df2 = 7, p-value = 0.1104
alternative hypothesis: true location is not equal to c(0,0)

```

Alternativ eignet sich die in Abschn. 7.4.3 vorgestellte Funktion `Anova()` aus dem `car` Paket. Für den multivariaten Test ist hier bei der Anwendung von `summary()` das Argument `multivariate=TRUE` zu setzen. Alle dabei ausgegebenen Teststatistiken sind äquivalent zum T^2 -Test, wenn (wie hier letztlich) ein multivariater Test für nur eine Gruppe vorliegt. Da das theoretische Zentroid unter H_0 gleich **0** ist, entfällt hier die Notwendigkeit, es zuvor von den Zeilenvektoren der Datenmatrix der Differenzvariablen abzuziehen.

```

> fitRBp   <- lm(DVw ~ 1)                      # Zw.-Gruppen Design
> intraRBp <- data.frame(IV=gl(P, 1))          # Intra-Gruppen Design
> library(car)                                 # für Anova()
> AnovaRBp <- Anova(fitRBp, idata=intraRBp, idesign=~IV)
> summary(AnovaRBp, multivariate=TRUE, univariate=FALSE)    # ...

```

12.7 Multivariate Varianzanalyse (MANOVA)

12.7.1 Einfaktorielle MANOVA

Die einfaktorielle multivariate Varianzanalyse prüft die in mehreren Bedingungen einer UV erhobenen Datenvektoren mehrerer gemeinsam normalverteilter Variablen mit identischen Kovarianzmatrizen daraufhin, ob sie mit der H_0 konsistent sind, dass ihre theoretischen Zentroide übereinstimmen. Der Test ist äquivalent zur univariaten einfaktoriellen Varianzanalyse, wenn nur die Daten einer AV vorliegen (vgl. Abschn. 7.3).

Zur Durchführung eignet sich wie bei Hotellings T^2 -Test für zwei Stichproben die `manova()` Funktion als Verallgemeinerung von `aov()`. Dabei ist der linke AV-Teil der Modellformel multivariat zu formulieren, also eine spaltenweise aus den Variablen zusammengestellte Matrix mit den Beobachtungsobjekten in den Zeilen zu übergeben. Als UV ist ein Faktor zu nennen, der für jede Zeile der AV-Matrix codiert, aus welcher Bedingung der Datenvektor stammt.

Im Test der von `manova()` durchgeführten Modellanpassung mit `summary()` können verschiedene Teststatistiken über das Argument `test` gewählt werden: Voreinstellung ist "Pillai" für die

Pillai-Bartlett-Spur, andere Optionen sind "Wilks" für Wilks' Λ , "Roy" für Roys Maximalwurzel und "Hotelling-Lawley" für die Hotelling-Lawley-Spur (vgl. Abschn. 12.9.7). Bei zwei Gruppen sind alle Teststatistiken äquivalent.

Das Beispiel aus Abschnitt 12.6.1 und 12.6.2 soll nun um die in einer dritten Bedingung erhobenen Daten der betrachteten beiden Variablen erweitert werden (für die manuelle Kontrolle vgl. Abschn. 12.9.9).

```
> mu31 <- c(1, -1)                      # Zentroid Zufallsvektoren 3. Bedingung

# Datenmatrix aus 3. Bedingung mit Variablen in den Spalten
> library(mvtnorm)                      # für rmvnorm()
> Y31 <- round(rmvnorm(Nj[3], mean=mu31, sigma=sigma))
> Ym1 <- rbind(Y11, Y21, Y31)      # Gesamt-Datenmatrix aller Bedingungen

# codiere für jede Zeile der Datenmatrix die zugehörige Bedingung
> IVman    <- factor(rep(1:3, Nj))
> manRes1 <- manova(Ym1 ~ IVman)
> summary(manRes1, test="Wilks")           # Wilks' Lambda
      Df   Wilks   approx F num Df  den Df   Pr(>F)
IVman     2  0.42011    15.199       4      112  5.89e-10 ***
Residuals 57

> summary(manRes1, test="Roy")              # Roys Maximalwurzel ...
> summary(manRes1, test="Pillai")            # Pillai-Bartlett-Spur ...
> summary(manRes1, test="Hotelling-Lawley") # Hotelling-Lawley-Spur ...
```

12.7.2 Zweifaktorielle MANOVA

Die zweifaktorielle multivariate Varianzanalyse ist wie die einfaktorielle MANOVA mit `manova()` durchzuführen, lediglich die Spezifikation der rechten UV-Seite der Modellformel erweitert sich um die zusätzlich zu berücksichtigenden Effekte.⁴² In der Rolle der AV auf der linken Seite der Formel steht weiterhin eine spaltenweise aus den Variablen zusammengesetzte Matrix mit den Beobachtungsobjekten in den Zeilen. Als UV können nun die Faktoren genannt werden, deren Stufen die Bedingungskombinationen festlegen, in denen die AVn erhoben wurden. Jeder Faktor gibt für jede Zeile der Datenmatrix an, aus welcher Bedingung bzgl. der durch ihn codierten UV der Datenvektor stammt (für die manuelle Kontrolle vgl. Abschn. 12.9.10).

```
# Daten aus den 3 Bedingungen der UV 1 in 2. Stufe der UV 2
> mu12 <- c(-1, 4)                      # Zentroid Zufallsvektoren 1. Bedingung
> mu22 <- c( 4, 8)                      # Zentroid Zufallsvektoren 2. Bedingung
> mu32 <- c( 4, 0)                      # Zentroid Zufallsvektoren 3. Bedingung
> library(mvtnorm)                      # für rmvnorm()
```

⁴² Auch bei der multivariaten zweifaktoriellen Varianzanalyse ist im Fall ungleicher Zellbesetzungen zu beachten, dass R in der Voreinstellung Quadratsummen vom Typ I berechnet (vgl. Abschn. 7.5.2, 12.9.6). `Manova()` aus dem `car` Paket erlaubt es, analog zur Verwendung von `Anova()` (vgl. Abschn. 7.4.3), Quadratsummen vom Typ II und III zu berechnen.

```

> Y12 <- round(rmvnorm(Nj[1], mu12, sigma))
> Y22 <- round(rmvnorm(Nj[2], mu22, sigma))
> Y32 <- round(rmvnorm(Nj[3], mu32, sigma))

# vollständige Datenmatrix aus den 3x2 Bedingungen (Variablen = Spalten)
> Ym2 <- rbind(Ym1, Y12, Y22, Y32)

# codiere für jede Zeile der Datenmatrix zugehörige Bedingung beider UVn
> IV1 <- rep(IVman, times=2)
> IV2 <- factor(rep(1:2, each=sum(Nj)))

> manRes2 <- manova(Ym2 ~ IV1*IV2)
> summary(manRes2, test="Pillai")                      # Pillai-Bartlett-Spur
> summary(manRes2, test="Pillai")
      Df    Pillai   approx F    num Df    den Df     Pr(>F)
IV1       2  0.72468     32.389      4      228 < 2.2e-16 ***
IV2       1  0.18107     12.493      2      113  1.254e-05 ***
IV1:IV2   2  0.15762      4.876      4      228  0.0008615 ***
Residuals 114

> summary(manRes2, test="Wilks")                      # Wilks' Lambda ...
> summary(manRes2, test="Roy")                        # Roy's Maximalwurzel ...
> summary(manRes2, test="Hotelling-Lawley")          # Hotelling-Lawley-Spur ...

```

12.8 Diskriminanzanalyse

Die Diskriminanzanalyse bezieht sich auf dieselbe Erhebungssituation wie die einfaktorielle MANOVA und teilt deren Voraussetzungen (vgl. Abschn. 12.7.1): Beobachtungsobjekte aus p Gruppen liefern Werte auf r quantitativen AVn Y_l . Diese Variablen seien in jeder Gruppe multinormalverteilt mit derselben invertierbaren Kovarianzmatrix, aber u. U. abweichenden Erwartungswertvektoren. Anlass zur Anwendung kann eine zuvor durchgeführte signifikante MANOVA sein, deren unspezifische Alternativhypothese offen lässt, wie genau sich die Gruppenzentroide unterscheiden.

Die Diskriminanzanalyse erzeugt $\min(p - 1, r)$ viele Linearkombinationen $LD = b_0 + b_1Y_1 + \dots + b_rY_r$ der ursprünglichen Variablen, auf denen sich die Gruppenunterschiede im folgenden Sinne besonders deutlich zeigen:⁴³ Die als *Diskriminanzfunktionen*, oder auch als *Fishers lineare Diskriminanten* bezeichneten LD sind unkorrelierte Variablen, deren jeweiliger F -Bruch aus der einfaktoriellen Varianzanalyse mit der Diskriminanten als AV sukzessive maximal ist. Hier zeigt sich eine Ähnlichkeit zur Hauptkomponentenanalyse (vgl. Abschn. 12.2), die die Gruppenzugehörigkeit der Objekte jedoch nicht berücksichtigt und neue unkorrelierte Variablen mit schrittweise maximaler Varianz bildet.

⁴³Lässt man auch quadratische Funktionen der ursprünglichen Variablen zu, ergibt sich die quadratische Diskriminanzanalyse. Sie wird mit `qda()` aus dem MASS Paket berechnet.

Die Diskriminanzanalyse lässt sich auch mit der Zielsetzung durchführen, Objekte anhand mehrerer Merkmale möglichst gut hinsichtlich eines bestimmten Kriteriums klassifizieren zu können. Hierfür wird zunächst ein Trainingsdatensatz benötigt, von dessen Objekten sowohl die Werte der diagnostischen Variablen als auch ihre Gruppenzugehörigkeit bekannt sind. Mit diesem Datensatz werden die Koeffizienten der Diskriminanzfunktionen bestimmt, die zur späteren Klassifikation anderer Objekte ohne bekannte Gruppenzugehörigkeit dienen.⁴⁴ Die lineare Diskriminanzanalyse lässt sich mit `lda()` aus dem MASS Paket durchführen.

```
> lda(formula=<Modellformel>, data=<Datensatz>, subset=<Indexvektor>,
+      CV=<Kreuzvalidierung>, prior=<Basiswahrscheinlichkeiten>,
+      method=<Kovarianzschätzung>)"
```

Das erste Argument ist eine Modellformel der Form $\langle UV \rangle \sim \langle AV \rangle$. Bei ihr ist abweichend von den bisher betrachteten linearen Modellen die links von der \sim stehende, vorherzusagende Variable ein Faktor der Gruppenzugehörigkeiten, während die quantitativen AVn die Rolle der Prädiktoren auf der rechten Seite einnehmen. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Das Argument `subset` erlaubt es, nur eine Auswahl der Fälle einfließen zu lassen, etwa wenn eine Trainingsstichprobe als zufällige Teilmenge eines Datensatzes gebildet werden soll. `subset` erwartet einen Indexvektor, der sich auf die Zeilen des Datensatzes bezieht.

In der Voreinstellung verwendet `lda()` die relativen Häufigkeiten der Gruppen als Maß für ihre Auftretenswahrscheinlichkeiten in der Population. Über das Argument `prior` lassen sich letztere auch explizit in Form eines Vektors in der Reihenfolge der Stufen von $\langle UV \rangle$ vorgeben. Setzt man das Argument `method="mve"`, verwendet `lda()` eine robuste Schätzung von Mittelwerten und Kovarianzmatrix.

Das Beispiel verwendet dieselben Daten wie die einfaktorielle MANOVA (vgl. Abschn. 12.7.1), wobei die ungleichen Gruppenhäufigkeiten zunächst als Indikator für ihre Wahrscheinlichkeiten dienen sollen.

```
> Ydf1 <- data.frame(IVman, DV1=Ym1[, 1], DV2=Ym1[, 2])
> library(MASS)                                     # für lda()
> (ldaRes <- lda(IVman ~ DV1 + DV2, data=Ydf1))
Call:
lda(IVman ~ DV1 + DV2, data = Ydf1)

Prior probabilities of groups:
      1         2         3 
0.2500000 0.4166667 0.3333333 

Group means:
      DV1     DV2
1 -3.266667 5.60
```

⁴⁴Für weitere Klassifikationsverfahren wie Varianten der Clusteranalyse, CART-Modelle oder *support vector machines* vgl. die Abschnitte *Cluster Analysis* (Leisch & Gruen, 2014), *Multivariate Statistics* (Hewson, 2014) und *Machine Learning & Statistical Learning* (Hothorn, 2014) der CRAN Task Views. Die logistische und multinomiale Regression (vgl. Abschn. 8.1, 8.3) lassen sich ebenfalls zur Klassifikation verwenden und besitzen weniger Verteilungsvoraussetzungen als die Diskriminanzanalyse.

```
2 2.360000 4.04
3 -0.350000 -0.40
```

Coefficients of linear discriminants:

	LD1	LD2
DV1	0.06710397	-0.27380306
DV2	-0.31861364	-0.07850291

Proportion of trace:

LD1	LD2
0.6327	0.3673

Die Ausgabe nennt unter der Überschrift **Prior probabilities of groups** die angenommenen Gruppenwahrscheinlichkeiten. Unter **Group means** folgt eine zeilenweise aus den Vektoren der Gruppenzentroide zusammengestellte Matrix, die in der von **lda()** zurückgegebenen Liste in der Komponente **means** enthalten ist. Die Koeffizienten b_l der Diskriminanzfunktionen finden sich spaltenweise unter **Coefficients of linear discriminants**, das Ergebnis speichert diese Matrix in der Komponente **scaling**. In der Ausgabe fehlen die für eine Interpretation der Ergebnisse meist nicht interessanten absoluten Terme b_0 der Linearkombinationen. Unter **Proportion of trace** lässt sich der Anteil der von jeder Diskriminanzfunktion aufgeklärten Varianz an der Gesamtvarianz zwischen den Gruppen im unten näher erläuterten Sinn ablesen.

Das Argument **CV=TRUE** (Voreinstellung ist **FALSE**) bewirkt eine Kreuzvalidierung, wobei gleichzeitig für jede Beobachtung und jede Gruppe die a-posteriori Wahrscheinlichkeit i. S. von Bayes berechnet wird, dass die Beobachtung zu einer Gruppe gehört. Die Matrix dieser Wahrscheinlichkeiten findet sich dann in der Komponente **posterior** der ausgegebenen Liste. Dagegen unterbleibt in diesem Fall die Berechnung der Koeffizienten für die Diskriminanzfunktionen.

```
> ldaP <- lda(IVman ~ DV1 + DV2, CV=TRUE, data=Ydf1)
> ldaP$posterior # Wahrscheinlichkeiten ...
```

Die aus der Regression bekannte **predict(***lda-Objekt***,** *Datensatz***)** Funktion (vgl. Abschn. 6.4) dient zur Vorhersage der Gruppenzugehörigkeiten auf Basis eines von **lda()** ausgegebenen Objekts. Dazu ist als zweites Argument ein Datensatz mit Variablen zu nennen, die dieselben Namen wie die AVn aus der ursprünglichen Analyse tragen. Die von **predict()** ausgegebene Liste enthält in der Komponente **x** die Diskriminanten selbst und in der Komponente **class** die vorhergesagte Kategorie. Die Güte der Vorhersage lässt sich für die Trainingsstichprobe etwa an der Konfusionsmatrix gemeinsamer Häufigkeiten von tatsächlichen und vorhergesagten Gruppenzugehörigkeiten ablesen (für weitere Maße der Übereinstimmung kategorialer Variablen vgl. Abschn. 10.2.6, 10.3.3).

```
> ldaPred <- predict(ldaRes, Ydf1) # Vorhersage für ursprüngliche Daten
> head(ldaPred$x) # Diskriminanten
   LD1          LD2
1 0.4166295  0.98818001
2 -2.6352990 -0.34445522
3 -2.7193092  1.37686604
4 -1.0591370  1.49557754
```

```

5 -3.7253478 -0.03235784
6  0.8694511  0.51907680

> head(ldaPred$class)          # Klassifikation
[1] 3 1 1 1 1 3
Levels: 1 2 3

# Kontingenztafel tatsächlicher und vorhergesagter Kategorien
> cTab <- table(IVman, ldaPred$class, dnn=c("IVman", "ldaPred"))
> addmargins(cTab)
      ldaPred
IVman   1   2   3  Sum
  1    9   3   3  15
  2    3  19   3  25
  3    1   6  13  20
Sum   13  28  19  60

> sum(diag(cTab)) / sum(cTab)      # Rate der korrekten Klassifikation
[1] 0.6833333

```

Die manuelle Kontrolle beruht auf den in Abschn. 12.9.9 berechneten Matrizen \mathbf{B} (SSP-Matrix der durch das zugehörige Gruppenzentroid ersetzen Daten) und \mathbf{W} (SSP-Matrix der Residuen). Die Koeffizientenvektoren der Diskriminanzfunktionen erhält man aus den Eigenvektoren von $\mathbf{W}^{-1}\mathbf{B}$. Diese werden dafür zum einen so umskaliert, dass die Residual-Quadratsumme der univariaten Varianzanalysen mit je einer Diskriminante als AV gleich $N - p$, die mittlere Residual-Quadratsumme also gleich 1 ist. Die F -Brüche dieser Varianzanalysen sind gleich den Eigenwerten von $\mathbf{W}^{-1}\mathbf{B}$, die mit dem Quotient der Freiheitsgrade der Quadratsummen innerhalb ($N - p$) und zwischen den Gruppen ($p - 1$) multipliziert wurden. Zum anderen werden die Diskriminanten so verschoben, dass ihr Mittelwert jeweils 0 beträgt. Der Anteil der Eigenwerte von $\mathbf{W}^{-1}\mathbf{B}$ an ihrer Summe, also an der Spur von $\mathbf{W}^{-1}\mathbf{B}$, wird von `lda()` unter *Proportion of trace* genannt.

```

> eigWinvB <- eigen(solve(WW) %*% BB)    # Eigenwerte, -vektoren W^-1 * B
> eigVec   <- eigWinvB$vectors            # Eigenvektoren
> eigVal   <- eigWinvB$values           # Eigenwerte
> p        <- nlevels(IVman)             # Anzahl Gruppen
> N        <- sum(Nj)                   # gesamt-N
> My       <- colMeans(Ym1)              # Mittelwerte Variablen

# Proportion of trace, alternativ: eigVal / sum(eigVal)
> eigVal / sum(diag(solve(WW) %*% BB))
[1] 0.6327184 0.3672816

# Skalierungsfaktoren für Eigenvektoren
> scl <- sqrt((N-p) / diag(t(eigVec) %*% WW %*% eigVec))
> b0  <- -scl * t(eigVec) %*% My        # absolute Terme b_0

```

```
# Skalierung der Eigenvektoren -> Matrix mit Koeffizienten b_k
> (bk <- eigVec %*% diag(scl))
      [,1]      [,2]
[1,]  0.06710397 -0.27380306
[2,] -0.31861364 -0.07850291

# prüfe, ob Diskriminanten mit Ergebnis von predict() übereinstimmen
> ld <- sweep(Ym1 %*% bk, 2, b0, "+") # Diskriminanten
> all.equal(ld, ldaPred$x, check.attributes=FALSE)
[1] TRUE

# univariate ANOVAs mit je einer Diskriminante als AV: SSw=N-p, MSw=1
> anova(lm(ld[, 1] ~ IVman))           # ANOVA mit 1. Diskriminante
Analysis of Variance Table
Response: ld1
Df Sum Sq Mean Sq F value    Pr(>F)
IVman   2 39.651 19.826 19.826 2.911e-07 ***
Residuals 57 57.000 1.000

> anova(lm(ld[, 2] ~ IVman))           # ANOVA mit 2. Diskriminante
Analysis of Variance Table
Response: ld2
Df Sum Sq Mean Sq F value    Pr(>F)
IVman   2 23.017 11.508 11.508 6.335e-05 ***
Residuals 57 57.000 1.0000

# F-Werte der ANOVAs aus Eigenwerten von W^-1 * B
> ((N-p) / (p-1)) * eigVal
[1] 19.82570 11.50846
```

Wurden der Diskriminanzanalyse gleiche Gruppenwahrscheinlichkeiten zugrunde gelegt, ergibt sich die vorhergesagte Gruppenzugehörigkeit für eine Beobachtung aus dem minimalen euklidischen Abstand zu den Gruppenzentroiden im durch die Diskriminanzfunktionen gebildeten Koordinatensystem: Dazu sind die Diskriminanten für alle Beobachtungen zu berechnen und die Gruppenmittelwerte der Trainingsstichprobe auf jeder Diskriminante zu bilden. Für die zu klassifizierende Beobachtung wird jene Gruppe als Vorhersage ausgewählt, deren Zentroid am nächsten an der Beobachtung liegt.

```
# Diskriminanzanalyse mit Annahme gleicher Gruppenwahrscheinlichkeiten
> priorP <- rep(1/nlevels(IVman), nlevels(IVman)) # gleiche Wkt.
> ldaEq <- lda(IVman ~ DV1 + DV2, prior=priorP, data=Ydf1)
> predEq <- predict(ldaEq, Ydf1)                  # Diskrim., Vorhersage
> LDmat <- predEq$x                                # Diskriminanten

# Datensatz der Gruppenzentroide
> ctrDf <- aggregate(cbind(LD1, LD2) ~ IVman, FUN=mean, data=LDmat)
> ctrLD <- data.matrix(ctrDf[, -1])                # Matrix Gruppenzentroide
```

```

# Matrizen der Differenzvektoren der Beobachtungen zu jedem Zentroid
> diffMat1 <- scale(LDmat, center=ctrLD[1, ], scale=FALSE) # zu Z1
> diffMat2 <- scale(LDmat, center=ctrLD[2, ], scale=FALSE) # zu Z2
> diffMat3 <- scale(LDmat, center=ctrLD[3, ], scale=FALSE) # zu Z3

# euklidische Distanzen als jeweilige Länge des Differenzvektors
> dst1   <- sqrt(diag(tcrossprod(diffMat1))) # zu Zentroid 1
> dst2   <- sqrt(diag(tcrossprod(diffMat2))) # zu Zentroid 2
> dst3   <- sqrt(diag(tcrossprod(diffMat3))) # zu Zentroid 3
> dstMat <- cbind(dst1, dst2, dst3)           # Matrix: alle Distanzen

# jede Zeile: identifizierte Spalte mit minimaler Distanz -> Vorhersage
> dstPred <- apply(dstMat, 1, which.min)
> head(dstPred)                                # Klassifikation
1 2 3 4 5 6
3 1 1 1 1 3

# prüfe auf Übereinstimmung mit Ergebnis von predict()
> all(dstPred == unclass(predEq$class))
[1] TRUE

```

12.9 Das allgemeine lineare Modell

Das allgemeine lineare Modell (ALM) liefert einen formalen Rahmen, mit dessen Hilfe sich u. a. lineare Regression (vgl. Abschn. 12.5), Varianzanalyse (vgl. Abschn. 12.7) und Kovarianzanalyse (vgl. Abschn. 7.8) auf dieselbe Weise formulieren und ihre Hypothesen testen lassen. Es integriert zudem die uni- wie multivariaten Varianten dieser Verfahren. Das ALM liegt implizit den Berechnungen von R zugrunde und wird dabei bisweilen für den Anwender sichtbar, weshalb hier die Grundidee skizziert werden soll. Weiterführende Darstellungen finden sich bei Andres (1996) und Mardia et al. (1980). Für die verwendeten Konzepte der linearen Algebra vgl. Abschn. 12.1.

12.9.1 Modell der multiplen linearen Regression

Das Modell der univariaten multiplen Regression geht von Beobachtungsobjekten $i = 1, \dots, n$ aus, die Daten y_i eines Kriteriums Y und Werte x_{ij} von p Prädiktoren X_j liefern, die jeweils in Vektoren \mathbf{y} und \mathbf{x}_j zusammengefasst werden (vgl. Abschn. 6.3).

$$\begin{aligned} E(y_i) &= \beta_0 + \beta_1 x_{i1} + \dots + \beta_j x_{ij} + \dots + \beta_p x_{ip} \\ E(\mathbf{y}) &= \beta_0 + \beta_1 \mathbf{x}_1 + \dots + \beta_j \mathbf{x}_j + \dots + \beta_p \mathbf{x}_p \end{aligned}$$

Dabei ist $E(\mathbf{y})$ der n -Vektor $(E(y_1), \dots, E(y_n))^t$ der Erwartungswerte von y_i . Von den skalaren Parametern β_0 (additive Konstante, absoluter Term) und β_j (theoretische Regressionsgewichte)

wird angenommen, dass sie für alle Beobachtungsobjekte identisch sind. Ferner sei vorausgesetzt, dass mehr Beobachtungen als Parameter vorhanden sind, hier also $n > p + 1$ gilt. In Matrix-Schreibweise lässt sich das Modell so formulieren:

$$E(\mathbf{y}) = \mathbf{1}\beta_0 + \mathbf{X}_p\beta_p = [\mathbf{1}|\mathbf{X}_p] \begin{pmatrix} \beta_0 \\ \beta_p \end{pmatrix} = \mathbf{X}\beta$$

Hier ist $\mathbf{1}$ der n -Vektor $(1, \dots, 1)^t$, β_p der p -Vektor $(\beta_1, \dots, \beta_p)^t$, β der $(p+1)$ -Vektor $(\beta_0, \beta_p^t)^t$ und \mathbf{X}_p die $(n \times p)$ -Matrix der spaltenweise zusammengestellten Vektoren \mathbf{x}_j . Die Prädiktoren sollen linear unabhängig sein, womit \mathbf{X}_p vollen Spaltenrang p besitzt. $\mathbf{X} = [\mathbf{1}|\mathbf{X}_p]$ ist die $(n \times (p+1))$ -Designmatrix, deren Spalten den Unterraum V mit Dimension $\text{Rang}(\mathbf{X}) = p+1$ aufspannen.⁴⁵ Das Modell lässt sich als Behauptung verstehen, dass $E(\mathbf{y})$ in V liegt. Für einen solchen modellverträglichen Vektor von Erwartungswerten existiert ein Parametervektor β , mit dem $E(\mathbf{y}) = \mathbf{X}\beta$ gilt.⁴⁶

Im ALM ergeben sich die personenweisen Erwartungswerte eines Kriteriums als lineare Funktion der Prädiktoren. Der Zusammenhang zwischen einer AV Y und einer UV X selbst muss im ALM dagegen nicht linear sein. Ein quadratischer Zusammenhang zwischen Y und X könnte etwa als Modell $E(\mathbf{y}) = \beta_0 + \beta_1 \mathbf{x} + \beta_2 \mathbf{x}^2$ formuliert werden – hier gäbe es bei einer UV X zwei Prädiktoren, nämlich X und X^2 .

Im Fall der multivariaten multiplen linearen Regression mit r Kriterien Y_l stellt man die Vektoren $E(\mathbf{y}_l)$ der Erwartungswerte der einzelnen Kriterien spaltenweise zu einer $(n \times r)$ -Matrix $E(\mathbf{Y})$ zusammen. Genauso werden die r vielen p -Vektoren $\beta_{p,l} = (\beta_{p,1l}, \dots, \beta_{p,jl}, \dots, \beta_{p,pl})^t$ der theoretischen Regressionsgewichte jeweils aus der Regression mit den Prädiktoren X_j und dem Kriterium Y_l spaltenweise zu einer $(p \times r)$ -Matrix \mathbf{B}_p zusammengefasst. Die r absoluten Terme $\beta_{0,l}$ bilden einen r -Vektor β_0 . Das Modell lautet dann:

$$E(\mathbf{Y}) = \beta_0 + \mathbf{X}_p \mathbf{B}_p = [\mathbf{1}|\mathbf{X}_p] \begin{pmatrix} \beta_0^t \\ \mathbf{B}_p \end{pmatrix} = \mathbf{X}\mathbf{B}$$

Die Designmatrix im multivariaten Fall stimmt mit jener im univariaten Fall überein. Die Modellparameter sind im univariaten Fall bei Kenntnis von \mathbf{X} und $E(\mathbf{y})$ als $\beta = \mathbf{X}^+ E(\mathbf{y})$ identifizierbar – der Lösung der *Normalengleichungen* (vgl. Abschn. 12.1.7). Dies gilt analog auch im multivariaten Fall mit $\mathbf{B} = \mathbf{X}^+ E(\mathbf{Y})$. Dabei ist $\mathbf{X}^+ = (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{X}^t$ die Pseudoinverse von \mathbf{X} , also $\mathbf{X}^+ \mathbf{X} = \mathbf{I}$ mit der $(n \times n)$ -Einheitsmatrix \mathbf{I} . Die Parameter sind gleich den Koordinaten des Vektors der Erwartungswerte, der orthogonal auf V projiziert wurde, bzgl. der durch \mathbf{X} definierten Basis (vgl. Abschn. 12.1.7).

⁴⁵ Die Designmatrix erhält man mit der Funktion `model.matrix()`, die als Argument ein mit `lm()` erstelltes Modell, oder auch nur die rechte Seite einer Modellformel akzeptiert (vgl. Abschn. 5.2 und Venables & Ripley, 2002, p. 144 ff.).

⁴⁶ Die \mathbf{x}_j sind feste Realisierungen eines Zufallsvektors, also *stochastische Prädiktoren*. Sie enthalten damit nicht alle möglichen Prädiktorwerte, sondern nur jeweils n viele. Man könnte daher auch vom Vektor $E(\mathbf{y}|\mathbf{X})$ der auf eine konkrete Designmatrix \mathbf{X} bedingten Erwartungswerte von \mathbf{y} sprechen, worauf hier aber verzichtet wird. Die \mathbf{x}_j müssen fehlerfrei sein, bei den x_{ij} muss es sich also um die wahren Prädiktorwerte handeln. Verzichtet man auf diese Annahme, werden lineare Strukturgleichungsmodelle zur Auswertung erforderlich (vgl. Abschn. 12.3, Fußnote 32).

Für eine univariate multiple Regression mit zwei Prädiktoren X_j und drei Beobachtungsobjekten, die Prädiktorwerte x_{ij} besitzen, ergibt sich folgendes Modell:

$$\begin{aligned} E(\mathbf{y}) &= \begin{pmatrix} E(y_1) \\ E(y_2) \\ E(y_3) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \beta_0 + \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{pmatrix}_{\mathbf{X}_p} \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}_{\boldsymbol{\beta}_p} \\ &= \begin{pmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ 1 & x_{31} & x_{32} \end{pmatrix}_{\mathbf{X}} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}_{\boldsymbol{\beta}} \\ &= \begin{pmatrix} \beta_0 + \beta_1 x_{11} + \beta_2 x_{12} \\ \beta_0 + \beta_1 x_{21} + \beta_2 x_{22} \\ \beta_0 + \beta_1 x_{31} + \beta_2 x_{32} \end{pmatrix} \end{aligned}$$

Im Fall einer moderierten Regression wird das Modell um zusätzliche Prädiktoren erweitert, die als Interaktionsterm gleich dem Produkt von Einzelprädiktoren sind (vgl. Abschn. 6.3.4). Hierfür kann man aus Produkten der Spalten von \mathbf{X}_p eine weitere Matrix \mathbf{X}_{pI} mit den neuen Prädiktoren erstellen, so dass die neue Designmatrix $\mathbf{X} = [\mathbf{1} | \mathbf{X}_p | \mathbf{X}_{pI}]$ ist. Der Parametervektor $\boldsymbol{\beta}$ ist entsprechend um passend viele Interaktionsparameter zu ergänzen.

12.9.2 Modell der einfaktoriellen Varianzanalyse

Analog zur Regression wird in der univariaten einfaktoriellen Varianzanalyse mit p unabhängigen Bedingungen (CR- p Design, vgl. Abschn. 7.3) zunächst die Zugehörigkeit zu jeder der p Faktorstufen zu einem eigenen dichotomen Prädiktor X_j^* , der als *Indikatorvariable* bezeichnet wird. Beobachtungsobjekte erhalten für ein X_j^* den Wert 1, wenn sie sich in der zugehörigen Bedingung j befinden, sonst 0.⁴⁷ Jedes Beobachtungsobjekt erhält also auf einem X_j^* die 1 und auf den verbleibenden $p - 1$ Indikatorvariablen die 0. Die spaltenweise aus den X_j^* zusammengestellte $(n \times p)$ -Matrix \mathbf{X}_p^* heißt *Inzidenzmatrix* und hat vollen Spaltenrang p . Die Komponenten des Vektors \mathbf{y} und die Zeilen der Inzidenzmatrix \mathbf{X}_p^* seien gemeinsam entsprechend der Gruppenzugehörigkeit geordnet.

Für die einfaktorielle Varianzanalyse mit drei Gruppen, zwei Personen pro Gruppe sowie den Parametern β_0 und β_j^* ergibt sich folgendes Modell. Beobachtungen aus unterschiedlichen

⁴⁷ Anders als in der Regression werden die Werte der Indikatorvariablen hier durch die Zuordnung von Beobachtungen zu Gruppen systematisch hergestellt, sind also keine stochastischen Prädiktoren.

Gruppen sind durch waagerechte Linien getrennt.

$$\begin{aligned}
 E(\mathbf{y}) &= \begin{pmatrix} \mu_1 \\ \mu_1 \\ \mu_2 \\ \mu_2 \\ \mu_3 \\ \mu_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}_1 \beta_0 + \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}_{\mathbf{X}_p^*} \begin{pmatrix} \beta_1^* \\ \beta_2^* \\ \beta_3^* \end{pmatrix}_{\beta_p^*} \\
 &= \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}_{\mathbf{X}^*} \begin{pmatrix} \beta_0 \\ \beta_1^* \\ \beta_2^* \\ \beta_3^* \end{pmatrix}_{\beta^*} = \begin{pmatrix} \beta_0 + \beta_1^* \\ \beta_0 + \beta_1^* \\ \beta_0 + \beta_2^* \\ \beta_0 + \beta_2^* \\ \beta_0 + \beta_3^* \\ \beta_0 + \beta_3^* \end{pmatrix}
 \end{aligned}$$

Der Vektor $\mathbf{1}$ ist die Summe der Spalten von \mathbf{X}_p^* , liegt also in ihrem Erzeugnis, weshalb die $(n \times (p+1))$ -Designmatrix $\mathbf{X}^* = [\mathbf{1} | \mathbf{X}_p^*]$ wie \mathbf{X}_p^* selbst nur Rang p besitzt. $(\mathbf{X}^*)^+$ ist nun nicht eindeutig bestimmt, da $(\mathbf{X}^*)^t \mathbf{X}^*$ nicht invertierbar ist. Die Parameter sind hier deswegen anders als in der Regression zunächst nicht identifizierbar. Aus diesem Grund verändert man das Modell, indem man eine Nebenbedingung $\mathbf{v}^t \beta_p^* = 0$ über die Parameter einführt und so deren frei variierbare Anzahl um 1 reduziert. Dabei ist \mathbf{v} ein p -Vektor, der eine Linearkombination der Parameter β_j^* festlegt, die 0 ist.⁴⁸

Es sei nun $\beta_{p-1} = (\beta_1, \dots, \beta_{p-1})^t$ der $(p-1)$ -Vektor der neuen, *reduzierten* Parameter, die auch *Kontraste* heißen. Weiter sei \mathbf{C} eine $(p \times (p-1))$ -Matrix, so dass die $(p \times p)$ -Matrix $[\mathbf{1} | \mathbf{C}]$ vollen Rang p besitzt und unter Erfüllung der Nebenbedingung $\mathbf{v}^t \beta_p^* = 0$ gilt:⁴⁹

$$\beta_p^* = \mathbf{C} \beta_{p-1}$$

\mathbf{C} wird *Kontrastmatrix* genannt und definiert für jede in den p Zeilen stehende Gruppenzugehörigkeit die zugehörigen Werte auf den neuen, nun $p-1$ Prädiktoren X_j , die mit den Spalten von \mathbf{C} korrespondieren. Die Nebenbedingung $\mathbf{v}^t \beta_p^* = 0$ ist erfüllt, wenn \mathbf{C} so gewählt wird, dass $\mathbf{v}^t \mathbf{C} = \mathbf{0}^t$ ist.⁵⁰ Dabei können für ein bestimmtes \mathbf{v} mehrere Wahlmöglichkeiten für \mathbf{C} bestehen.

Da \mathbf{C} vollen Spaltenrang $p-1$ besitzt, ist ihre $((p-1) \times p)$ -Links-Pseudoinverse mit $\mathbf{C}^+ = (\mathbf{C}^t \mathbf{C})^{-1} \mathbf{C}^t$ eindeutig definiert, und es gilt mit $\mathbf{C}^+ \mathbf{C} = \mathbf{I}$:

$$\beta_{p-1} = \mathbf{C}^+ \mathbf{C} \beta_{p-1} = \mathbf{C}^+ \beta_p^*$$

⁴⁸ Alternativ kann auch der Parameter $\beta_0 = 0$ gesetzt werden, womit $\mathbf{X}^* = \mathbf{X}_p^*$ ist. Diese Möglichkeit zur Parametrisierung soll hier nicht weiter verfolgt werden, um das Modell wie jenes der Regression formulieren zu können (vgl. Fußnote 53).

⁴⁹ Für die Parameterschätzungen gilt dies analog (vgl. Abschn. 12.9.4, Fußnote 59).

⁵⁰ Denn dann gilt $\mathbf{v}^t \beta_p^* = \mathbf{v}^t \mathbf{C} \beta_{p-1} = \mathbf{0}^t \beta_{p-1} = 0$. \mathbf{v} steht senkrecht auf den Spalten von \mathbf{C} , ist also eine Basis des orthogonalen Komplements des von den Spalten von \mathbf{C} aufgespannten Unterraums. Mit anderen Worten ist \mathbf{v} wegen $\mathbf{0} = (\mathbf{v}^t \mathbf{C})^t = \mathbf{C}^t \mathbf{v}$ eine Basis des Kerns von \mathbf{C}^t .

Die Kontrastmatrix \mathbf{C} lässt sich also auch so deuten, dass ihre Pseudoinverse \mathbf{C}^+ den Vektor β_p^* der ursprünglichen Parameter auf den Vektor β_{p-1} der Kontraste im Unterraum abbildet, in dem β_p^* mit der gewählten Nebenbedingung frei variieren kann. Anders gesagt ergeben sich die Kontraste β_{p-1} als Linearkombinationen der ursprünglichen Parameter β_p^* , wobei \mathbf{C}^+ die zeilenweise aus den Koeffizientenvektoren zusammengestellte Matrix mit Rang $p - 1$ ist. Die Zeilen von \mathbf{C}^+ bilden daher eine Basis des Raums der Koeffizientenvektoren unter der gewählten Nebenbedingung \mathbf{v} . Dabei erfüllt $\mathbf{C}^+ \mathbf{v} = \mathbf{0}$,⁵¹ \mathbf{v} ist also eine Basis des Kerns der Pseudoinversen \mathbf{C}^+ von \mathbf{C} . Das reduzierte Modell lautet damit:

$$E(\mathbf{y}) = \mathbf{1}\beta_0 + \mathbf{X}_p^* \beta_p^* = \mathbf{1}\beta_0 + \mathbf{X}_p^* \mathbf{C} \beta_{p-1} = [\mathbf{1} | \mathbf{X}_{p-1}] \begin{pmatrix} \beta_0 \\ \beta_{p-1} \end{pmatrix} = \mathbf{X}\beta$$

Hier ist $\mathbf{X}_{p-1} = \mathbf{X}_p^* \mathbf{C}$ die reduzierte $(n \times (p-1))$ -Inzidenzmatrix und $\mathbf{X} = [\mathbf{1} | \mathbf{X}_{p-1}]$ die reduzierte $(n \times p)$ -Designmatrix mit vollem Spaltenrang p , wobei $n > p$ vorausgesetzt wird. Der p -Vektor β der Kontraste ist so wie im Fall der Regression über $\mathbf{X}^+ E(\mathbf{y})$ identifizierbar.

Die inhaltliche Bedeutung der Kontraste in β hängt von der Wahl der Nebenbedingung \mathbf{v} und der Kontrastmatrix \mathbf{C} ab. R verwendet in der Voreinstellung die *Dummy-Codierung* (auch *Treatment-Kontraste* genannt), bei denen die Indikatorvariablen X_j^* zunächst erhalten bleiben. In der Voreinstellung wird dann der ursprüngliche, zur ersten Gruppe gehörende Parameter $\beta_1^* = 0$ gesetzt.⁵² Hier ist die Nebenbedingung also $\beta_1^* = (1, 0, \dots, 0)\beta_p^* = 0$, d. h. $\mathbf{v} = (1, 0, \dots, 0)^t$. `contr.treatment(Anzahl, base=Referenzgruppe)` gibt die Matrix \mathbf{C}_t für Treatment-Kontraste aus. Als Argument ist dabei die Anzahl der Gruppen p sowie optional die Nummer der Referenzstufe zu nennen – in der Voreinstellung die Stufe 1. Die Spalten von \mathbf{C}_t sind paarweise orthogonal und besitzen die Länge 1 ($\mathbf{C}_t^t \mathbf{C}_t = \mathbf{I}$), hier gilt also $\mathbf{C}_t^+ = \mathbf{C}_t^t$.

```
> contr.treatment(4)                                # Treatment-Kontraste für 4 Gruppen
   2 3 4
1 0 0 0
2 1 0 0
3 0 1 0
4 0 0 1
```

Treatment-Kontraste bewirken, dass die reduzierte Inzidenzmatrix \mathbf{X}_{p-1} durch Streichen der ersten Spalte von \mathbf{X}_p^* entsteht. Die Bezeichnung dieses Codierschemas leitet sich aus der Situation ab, dass die erste Faktorstufe eine Kontrollgruppe darstellt, während die übrigen zu Treatment-Gruppen gehören. Die Parameter können dann als Wirkung der Stufe j i. S. der Differenz zur Kontrollgruppe verstanden werden: Für die Mitglieder der ersten Gruppe ist $E(y_i) = \mu_1 = \beta_0$, da für diese Gruppe alle verbleibenden $X_j = 0$ sind (mit $j = 2, \dots, p$). Für Mitglieder jeder übrigen Gruppe j erhält man $E(y_i) = \mu_j = \beta_0 + \beta_j = \mu_1 + \beta_j$, da dann $X_j = 1$ ist. Damit ist $\beta_j = \mu_j - \mu_1$ gleich der Differenz des Erwartungswertes der Gruppe j zum Erwartungswert der Referenzgruppe.⁵³

⁵¹ $\mathbf{C}^+ \mathbf{v} = (\mathbf{v}^t \mathbf{C}^{+t})^t = (\mathbf{v}^t \mathbf{C} (\mathbf{C}^t \mathbf{C})^{-1})^t = (\mathbf{0}^t (\mathbf{C}^t \mathbf{C})^{-1})^t = \mathbf{0}$.

⁵² Die in einem linearen Modell mit kategorialen Variablen von R weggelassene Gruppe ist die erste Stufe von `levels(Faktor)`.

⁵³ Die in Fußnote 48 erwähnte Möglichkeit der Parametrisierung führt zum *cell means* Modell, bei dem $\mathbf{X} = \mathbf{X}^* = \mathbf{X}_p^*$ gilt und die Parameter β_j^* direkt die Bedeutung der Gruppenerwartungswerte μ_j erhalten.

Für $p = 3$, zwei Personen pro Gruppe und Treatment-Kontraste ergibt sich folgendes Modell mit den reduzierten Parametern β_0 und β_j :

$$\begin{aligned} E(\mathbf{y}) &= \begin{pmatrix} \mu_1 \\ \mu_1 \\ \mu_2 \\ \mu_2 \\ \mu_3 \\ \mu_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}_1 \beta_0 + \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}_{\mathbf{X}_p^*} \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}_{\mathbf{C}_t} \begin{pmatrix} \beta_2 \\ \beta_3 \end{pmatrix}_{\beta_{p-1}} \\ &= \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}_{\mathbf{X}} \begin{pmatrix} \beta_0 \\ \beta_2 \\ \beta_3 \end{pmatrix}_{\boldsymbol{\beta}} = \begin{pmatrix} \beta_0 \\ \beta_0 \\ \frac{\beta_0 + \beta_2}{\beta_0 + \beta_2} \\ \frac{\beta_0 + \beta_2}{\beta_0 + \beta_3} \\ \frac{\beta_0}{\beta_0 + \beta_3} \\ \frac{\beta_0}{\beta_0 + \beta_3} \end{pmatrix} \end{aligned}$$

In der mit `options("contrasts")` einsehbaren Voreinstellung verwendet R Treatment-Kontraste, wenn ungeordnete kategoriale Prädiktoren in einem linearen Modell vorhanden sind, und polynomiale Kontraste für ordinale Prädiktoren (für andere Möglichkeiten vgl. `?contrasts`). Deren Stufen werden dabei als gleichabständig vorausgesetzt.

```
> options("contrasts") # eingestelltes Codierschema
$contrasts
  unordered      ordered
"contr.treatment"  "contr.poly"
```

In der einfaktoriellen Varianzanalyse ist die Konzeption der Parameter jedoch oft eine andere. Hier sind dies die Effektgrößen $\alpha_j = \mu_j - \mu$, also die jeweilige Differenz der Gruppenerwartungswerte zum (ungewichteten, also gleichgewichteten) mittleren Erwartungswert $\mu = \frac{1}{p} \sum_j \mu_j$. Die Summe der so definierten Effektgrößen α_j in der Rolle der Parameter β_j^* ist 0. Die Nebenbedingung lautet hier also $\sum_j \beta_j^* = \mathbf{1}^t \boldsymbol{\beta}_p^* = 0$, d. h. $\mathbf{v} = \mathbf{1}$.

Die genannte Parametrisierung lässt sich über die ungewichtete *Effektcodierung* ausdrücken:⁵⁴ Hierfür wird zunächst die Zugehörigkeit zu jeder Faktorstufe zu einem separaten Prädiktor X_j , der die Werte $-1, 0$ und 1 annehmen kann. Beobachtungsobjekte aus der Gruppe j (mit $j = 1, \dots, p-1$) erhalten für X_j den Wert 1 , sonst 0 . Beobachtungsobjekte aus der Gruppe p erhalten auf allen X_j den Wert -1 . Zur Beseitigung der Redundanz wird dann der ursprüngliche, zur letzten Stufe gehörende Parameter β_p^* aus dem Modell gestrichen.⁵⁵ `contr.sum(Anzahl` ↴

⁵⁴ Ein andere Wahl für \mathbf{C} unter der Nebenbedingung $\mathbf{v} = \mathbf{1}$ ist die Helmert-Codierung mit paarweise orthogonalen Spalten von $[1|\mathbf{C}]$ (vgl. `?contr.helmert`). Die Parameter haben dann jedoch eine andere Bedeutung.

⁵⁵ Alternativ ließe sich $\mu = \sum_j \frac{n_j}{n} \mu_j$ als mit den anteiligen Zellbesetzungen $\frac{n_j}{n}$ gewichtetes Mittel der μ_j definieren.

Die zugehörige Nebenbedingung für die β_j^* i. S. der α_j lautet dann $\sum_j \frac{n_j}{n} \beta_j^* = 0$, d. h. $\mathbf{v} = (\frac{n_1}{n}, \dots, \frac{n_p}{n})^t$. Diese Parametrisierung lässt sich mit der gewichteten Effektcodierung umsetzen, die zunächst der ungewichteten gleicht. In der letzten Zeile der Matrix \mathbf{C} erhalten hier jedoch Mitglieder der Gruppe p für die X_j nicht den Wert -1 , sondern $-\frac{n_j}{n_p}$.

`→Gruppen()`) gibt die Matrix C_e für die Effektcodierung aus, wobei als Argument die Anzahl der Gruppen p zu nennen ist.

```
> contr.sum(4)                                # Effektcodierung für 4-stufigen Faktor
[ ,1] [ ,2] [ ,3]
1     1     0     0
2     0     1     0
3     0     0     1
4    -1    -1    -1
```

Mit der ungewichteten Effektcodierung erhält der Parameter β_0 die Bedeutung des ungewichteten mittleren Erwartungswertes $\mu = \frac{1}{p} \sum_j \mu_j$ und die β_j die Bedeutung der ersten $p-1$ Effektgrößen α_j . Für Mitglieder der ersten $p-1$ Gruppen ist nämlich $E(y_i) = \mu_j = \beta_0 + \beta_j$, für Mitglieder der Gruppe p gilt $E(y_i) = \mu_p = \beta_0 - (\beta_1 + \dots + \beta_{p-1})$. Dabei stellt $\beta_1 + \dots + \beta_{p-1}$ die Abweichung $\alpha_p = \mu_p - \mu$ dar, weil sich die Abweichungen der Gruppenerwartungswerte vom mittleren Erwartungswert über alle Gruppen zu 0 summieren müssen. In jeder Komponente ist somit $E(y_i) = \mu + \alpha_j$.

Für $p = 3$, zwei Personen pro Gruppe und Effektcodierung ergibt sich folgendes Modell mit den reduzierten Parametern β_0 und β_j :

$$E(\mathbf{y}) = \begin{pmatrix} \mu_1 \\ \mu_1 \\ \mu_2 \\ \mu_2 \\ \mu_3 \\ \mu_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}_1 \beta_0 + \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}_{X_p^*} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix}_{C_e} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_{p-1} \end{pmatrix}_{\beta_{p-1}}$$

$$= \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ \hline 1 & 0 & 1 \\ 1 & 0 & 1 \\ \hline 1 & -1 & -1 \\ 1 & -1 & -1 \end{pmatrix}_X \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}_{\beta} = \begin{pmatrix} \beta_0 + \beta_1 \\ \beta_0 + \beta_1 \\ \hline \beta_0 + \beta_2 \\ \beta_0 + \beta_2 \\ \hline \beta_0 - (\beta_1 + \beta_2) \\ \beta_0 - (\beta_1 + \beta_2) \end{pmatrix}$$

Soll R bei ungeordneten kategorialen Prädiktoren in diesem Sinne jede Gruppe nicht wie bei Treatment-Kontrasten mit der Referenzstufe, sondern durch die Effektcodierung generell mit dem Gesamtmittel verglichen, ist dies mit `options()` einzustellen.

```
# gehe zur Effektcodierung für ungeordnete Faktoren über
> options(contrasts=c("contr.sum", "contr.poly"))
```

```
# wechsle zurück zu Treatment-Kontrasten
> options(contrasts=c("contr.treatment", "contr.poly"))
```

Neben dieser Grundeinstellung existiert auch die Möglichkeit, das Codierschema für einen Faktor mit `C()` direkt festzulegen.

```
> C(object=<Faktor>, contr=<Kontrastmatrix>)
```

Ihr erstes Argument ist ein Faktor. Für das Argument `contr` ist entweder eine selbst erstellte Kontrastmatrix C oder eine Funktion wie etwa `contr.sum` zu übergeben, die eine passende Kontrastmatrix erzeugt. C wird als Attribut des von `C()` zurückgegebenen Faktors gespeichert und automatisch von Funktionen wie `lm()` oder `aov()` verwendet, wenn der neue Faktor Teil der Modellformel ist.

```
> IV <- gl(3, 5) # Faktor: 3 Gruppen à 5 Personen
> (IVe <- C(IV, contr.sum)) # Effektcodierung für Faktor IVe
[1] 1 1 1 1 1 2 2 2 2 3 3 3 3 3
attr(,"contrasts")
 [,1] [,2]
1     1   0
2     0   1
3    -1  -1
Levels: 1 2 3
```

`contrasts(<Faktor>)` gibt die passende Kontrastmatrix C für die Stufen des übergebenen Faktors aus. Ist mit dem Faktor keine Kontrastmatrix in Form eines Attributs fest verbunden, wird hierfür die Grundeinstellung verwendet, die mit `options("contrasts")` einsehbar ist.

```
> contrasts(IV) # Kontrastmatrix C nach Grundeinstellung
 2 3
1 0 0
2 1 0
3 0 1
```

Schließlich lässt sich das Codierschema temporär auch direkt beim Aufruf der `lm()` Funktion angeben, indem ihr Argument `contrasts` verwendet wird. Das Argument erwartet eine Liste, deren Komponenten als Namen die Bezeichnungen der Faktoren besitzen, die in der Modellformel auftauchen. Die Komponente selbst ist dann eine Kontrastfunktion wie etwa `contr.sum`.

```
> DV <- rnorm(15)
> lm(DV ~ IV, contrasts=list(IV=contr.treatment)) # ...
```

12.9.3 Modell der zweifaktoriellen Varianzanalyse

In der univariaten zweifaktoriellen Varianzanalyse mit p Stufen der ersten und q Stufen der zweiten UV (CRF- pq Design, vgl. Abschn. 7.5) ist zunächst für jeden der beiden Haupteffekte eine Inzidenzmatrix analog zum einfaktoriellen Fall zu bilden. Dies sollen hier die $(n \times p)$ -Matrix \mathbf{X}_1^* für die erste UV und die $(n \times q)$ -Matrix \mathbf{X}_2^* für die zweite UV sein. Der zugehörige p -Vektor der Parameter für die erste UV sei $\beta_1^* = (\beta_{1,1}^*, \dots, \beta_{1,p}^*)^t$, der q -Vektor der Parameter für die zweite UV sei $\beta_2^* = (\beta_{2,1}^*, \dots, \beta_{2,q}^*)^t$.

Die $(n \times (p \cdot q))$ -Inzidenzmatrix $\mathbf{X}_{1 \times 2}^*$ für den Interaktionseffekt wird als spaltenweise Zusammenstellung aller $p \cdot q$ möglichen Produkte der Spalten von \mathbf{X}_1^* und \mathbf{X}_2^* gebildet (vgl. Abschn. 6.3.4). Der zugehörige $(p \cdot q)$ -Vektor der passend geordneten Parameter sei $\beta_{1 \times 2}^* = (\beta_{1 \times 2,11}^*, \dots, \beta_{1 \times 2,pq}^*)^t$.

Die $(n \times (p + q + p \cdot q))$ -Gesamt-Inzidenzmatrix ist dann $[\mathbf{X}_1^* | \mathbf{X}_2^* | \mathbf{X}_{1 \times 2}^*]$ und der zugehörige $(p + q + p \cdot q)$ -Vektor aller genannten Parameter $(\beta_1^{*t}, \beta_2^{*t}, \beta_{1 \times 2}^{*t})^t$. Entsprechend ist die ursprüngliche $(n \times (p + q + p \cdot q + 1))$ -Designmatrix $\mathbf{X}^* = [\mathbf{1} | \mathbf{X}_1^* | \mathbf{X}_2^* | \mathbf{X}_{1 \times 2}^*]$ und das Modell analog zur einfaktoriellen Situation:

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_1^* | \mathbf{X}_2^* | \mathbf{X}_{1 \times 2}^*] \begin{pmatrix} \beta_0 \\ \beta_1^* \\ \beta_2^* \\ \beta_{1 \times 2}^* \end{pmatrix} = \mathbf{X}^* \beta^*$$

Für die zweifaktorielle Varianzanalyse mit $p = 3$, $q = 2$ und einer Person pro Gruppe ergibt sich folgendes Modell:

$$\begin{aligned} E(\mathbf{y}) &= \begin{pmatrix} \mu_{11} \\ \mu_{21} \\ \mu_{31} \\ \mu_{12} \\ \mu_{22} \\ \mu_{32} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}_{\mathbf{X}^*} \begin{pmatrix} \beta_0 \\ \beta_{1,1}^* \\ \beta_{1,2}^* \\ \beta_{1,3}^* \\ \beta_{2,1}^* \\ \beta_{2,2}^* \\ \beta_{1 \times 2,11}^* \\ \beta_{1 \times 2,21}^* \\ \beta_{1 \times 2,31}^* \\ \beta_{1 \times 2,12}^* \\ \beta_{1 \times 2,22}^* \\ \beta_{1 \times 2,32}^* \end{pmatrix}_{\beta^*} \\ &= \begin{pmatrix} \beta_0 + \beta_{1,1}^* + \beta_{2,1}^* + \beta_{1 \times 2,11}^* \\ \beta_0 + \beta_{1,2}^* + \beta_{2,1}^* + \beta_{1 \times 2,21}^* \\ \beta_0 + \beta_{1,3}^* + \beta_{2,1}^* + \beta_{1 \times 2,31}^* \\ \beta_0 + \beta_{1,1}^* + \beta_{2,2}^* + \beta_{1 \times 2,12}^* \\ \beta_0 + \beta_{1,2}^* + \beta_{2,2}^* + \beta_{1 \times 2,22}^* \\ \beta_0 + \beta_{1,3}^* + \beta_{2,2}^* + \beta_{1 \times 2,32}^* \end{pmatrix} \end{aligned}$$

Wie in der einfaktoriellen Varianzanalyse besitzt \mathbf{X}^* nicht vollen Spaltenrang (sondern Rang $(p - 1) + (q - 1) + (p - 1) \cdot (q - 1) + 1 = p \cdot q$), weshalb die Parameter nicht identifizierbar sind. Das Modell muss deshalb durch separate Nebenbedingungen an die Parameter der Haupteffekte und Interaktion wieder in eines mit weniger Parametern überführt werden. Zunächst seien dafür $\mathbf{v}_1^t \beta_1^* = 0$ und $\mathbf{v}_2^t \beta_2^* = 0$ die Nebenbedingungen für die Parameter der Haupteffekte, deren frei variierbare Anzahl sich dadurch auf $p - 1$ bzw. $q - 1$ reduziert. Weiter bezeichne $\mathbf{B}_{1 \times 2}^*$ die in einer $(p \times q)$ -Matrix angeordneten ursprünglichen Parameter $\beta_{1 \times 2}^*$ der Interaktion. Die Nebenbedingung für diese Parameter lässt sich dann als $\mathbf{v}_1^t \mathbf{B}_{1 \times 2}^* = \mathbf{0}^t$ und $\mathbf{B}_{1 \times 2}^* \mathbf{v}_2 = \mathbf{0}$ schreiben, wodurch deren frei variierbare Anzahl nunmehr $(p - 1) \cdot (q - 1)$ beträgt.

Analog zum einfaktoriellen Fall ist nun zunächst die $(p \times (p - 1))$ -Kontrastmatrix \mathbf{C}_1 für die $p - 1$ Kontraste der ersten UV im Vektor β_1 nach demselben Schema zu bilden wie die $(q \times (q - 1))$ -Kontrastmatrix \mathbf{C}_2 für die $q - 1$ Kontraste der zweiten UV im Vektor β_2 . Der Rang von $[1|\mathbf{C}_1]$ sei also p , der Rang von $[1|\mathbf{C}_2]$ sei q , und es gelte $\mathbf{v}_1^t \mathbf{C}_1 = \mathbf{0}$ ebenso wie $\mathbf{v}_2^t \mathbf{C}_2 = \mathbf{0}$. Die $((p \cdot q) \times (p - 1) \cdot (q - 1))$ -Kontrastmatrix $\mathbf{C}_{1 \times 2}$ für die $(p - 1) \cdot (q - 1)$ Kontraste der Interaktion im Vektor $\beta_{1 \times 2}$ erhält man als Kronecker-Produkt \otimes von \mathbf{C}_1 und \mathbf{C}_2 .⁵⁶ Der gemeinsame Vektor der reduzierten Parameter sei schließlich $\beta = (\beta_0, \beta_1^t, \beta_2^t, \beta_{1 \times 2}^t)^t$.

Der Zusammenhang zwischen den Kontrasten und den ursprünglichen Parametern, die den genannten Nebenbedingungen genügen, ist nun für jeden der drei Effekte wie im einfaktoriellen Fall. Der entsprechende Zusammenhang für die Parameter der Interaktion lässt sich dabei auf zwei verschiedene Arten formulieren – einmal mit $\beta_{1 \times 2}^*$ und einmal mit $\mathbf{B}_{1 \times 2}^*$.

$$\begin{aligned}\beta_1^* &= \mathbf{C}_1 \beta_1 \\ \beta_2^* &= \mathbf{C}_2 \beta_2 \\ \beta_{1 \times 2}^* &= \mathbf{C}_{1 \times 2} \beta_{1 \times 2} = (\mathbf{C}_2 \otimes \mathbf{C}_1) \beta_{1 \times 2} \\ \mathbf{B}_{1 \times 2}^* &= \mathbf{C}_1 \beta_{1 \times 2} \mathbf{C}_2^t\end{aligned}$$

Aus dem Produkt jeweils einer ursprünglichen Inzidenzmatrix mit der zugehörigen Kontrastmatrix berechnen sich die zugehörigen Inzidenzmatrizen für die Kontraste: $\mathbf{X}_1 = \mathbf{X}_1^* \mathbf{C}_1$ für β_1 , $\mathbf{X}_2 = \mathbf{X}_2^* \mathbf{C}_2$ für β_2 und $\mathbf{X}_{1 \times 2} = \mathbf{X}_{1 \times 2}^* \mathbf{C}_{1 \times 2}$ für $\beta_{1 \times 2}$. Dabei kann $\mathbf{X}_{1 \times 2}$ äquivalent auch nach demselben Prinzip wie $\mathbf{X}_{1 \times 2}^*$ gebildet werden: Die $(p - 1) \cdot (q - 1)$ paarweisen Produkte der Spalten von \mathbf{X}_1 und \mathbf{X}_2 werden dazu spaltenweise zu $\mathbf{X}_{1 \times 2}$ zusammengestellt. Die reduzierte $(n \times p \cdot q)$ -Designmatrix mit vollem Spaltenrang ist $\mathbf{X} = [1|\mathbf{X}_1|\mathbf{X}_2|\mathbf{X}_{1 \times 2}]$, wobei $n > p \cdot q$ vorausgesetzt wird. Das Modell mit identifizierbaren Parametern lautet damit wieder analog zur einfaktoriellen Situation:

$$E(\mathbf{y}) = [1|\mathbf{X}_1|\mathbf{X}_2|\mathbf{X}_{1 \times 2}] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_{1 \times 2} \end{pmatrix} = \mathbf{X} \beta$$

Der $p \cdot q$ -Vektor β der Kontraste ist so wie im einfaktoriellen Fall über $\mathbf{X}^+ E(\mathbf{y})$ identifizierbar.

Die inhaltliche Bedeutung der Kontraste in β hängt wie im einfaktoriellen Fall von der Wahl der Nebenbedingungen und Kontrastmatrizen ab. Voreinstellung in R sind Treatment-Kontraste: Für sie ergibt sich hier $\mathbf{v}_1 = (1, 0, \dots, 0)^t$ und $\mathbf{v}_2 = (1, 0, \dots, 0)^t$, d. h. es gilt $\beta_{1,1}^* = 0, \beta_{2,1}^* = 0$ sowie $\beta_{1 \times 2,1k}^* = 0$ für alle k wie auch $\beta_{1 \times 2,j1}^* = 0$ für alle j . In der ersten Zeile und Spalte von $\mathbf{B}_{1 \times 2}^*$ sind also alle Einträge 0. Mit Treatment-Kontrasten erhält man für die zweifaktorielle

⁵⁶Hierbei ist die Reihenfolge relevant, mit denen man die Spalten von $\mathbf{X}_{1 \times 2}^*$ und entsprechend die Parameter in $\beta_{1 \times 2}^*$ ordnet, die zu den Kombinationen jk der Faktorstufen der UV 1 und UV 2 gehören: Variiert (wie hier) der erste Index j schnell und der zweite Index k langsam ($\beta_{1 \times 2}^* = (\beta_{11}^*, \beta_{21}^*, \beta_{31}^*, \beta_{12}^*, \beta_{22}^*, \beta_{32}^*)^t$), gilt $\mathbf{C}_{1 \times 2} = \mathbf{C}_2 \otimes \mathbf{C}_1$. Dies ist die Voreinstellung in R, die sich etwa an der Ausgabe von `interaction((UV1), (UV2))` zeigt (vgl. Abschn. 2.6.2). Variiert dagegen j langsam und k schnell ($\beta_{1 \times 2}^* = (\beta_{11}^*, \beta_{12}^*, \beta_{21}^*, \beta_{22}^*, \beta_{31}^*, \beta_{32}^*)^t$), ist $\mathbf{C}_{1 \times 2} = \mathbf{C}_1 \otimes \mathbf{C}_2$ zu setzen.

Varianzanalyse mit $p = 3$, $q = 2$ und einer Person pro Gruppe folgende Kontrastmatrizen \mathbf{C}_t :

$$\mathbf{C}_{t1} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \mathbf{C}_{t2} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \mathbf{C}_{t1 \times 2} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Die ursprünglichen Parameter erhalten mit Treatment-Kontrasten folgende Bedeutung: $\beta_0 = \mu_{11}$, $\beta_{1,j}^* = \mu_{j1} - \mu_{11}$, $\beta_{2,k}^* = \mu_{1k} - \mu_{11}$, $\beta_{1 \times 2,jk}^* = (\mu_{jk} - \mu_{j1}) - (\mu_{1k} - \mu_{11})$. Das zugehörige Modell mit reduzierter Designmatrix \mathbf{X} und dem reduzierten Parametervektor $\boldsymbol{\beta}$ lautet:

$$E(\mathbf{y}) = \begin{pmatrix} \mu_{11} \\ \mu_{21} \\ \mu_{31} \\ \mu_{12} \\ \mu_{22} \\ \mu_{32} \end{pmatrix} = \left(\begin{array}{c|cc|c|cc} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \end{array} \right)_{\mathbf{X}} \begin{pmatrix} \beta_0 \\ \beta_{1,2} \\ \beta_{1,3} \\ \beta_{2,2} \\ \beta_{1 \times 2,22} \\ \beta_{1 \times 2,32} \end{pmatrix}_{\boldsymbol{\beta}}$$

$$= \begin{pmatrix} \beta_0 \\ \beta_0 + \beta_{1,2} \\ \beta_0 + \beta_{1,3} \\ \beta_0 + \beta_{2,2} \\ \beta_0 + \beta_{1,2} + \beta_{2,2} + \beta_{1 \times 2,22} \\ \beta_0 + \beta_{1,3} + \beta_{2,2} + \beta_{1 \times 2,32} \end{pmatrix}$$

Mit der ungewichteten Effektcodierung ergibt sich hier für die Nebenbedingungen $\mathbf{v}_1 = \mathbf{1}$ und $\mathbf{v}_2 = \mathbf{1}$, d.h. es gilt $\sum_j \beta_{1,j}^* = 0$, $\sum_k \beta_{2,k}^* = 0$ sowie $\sum_j \beta_{1 \times 2,jk}^* = 0$ für alle k wie auch $\sum_k \beta_{1 \times 2,jk}^* = 0$ für alle j (alle Zeilen- und Spaltensummen von $\mathbf{B}_{1 \times 2}^*$ sind 0). Für die zweifaktorielle Varianzanalyse mit $p = 3$, $q = 2$ und einer Person pro Gruppe erhält man mit Effektcodierung folgende Kontrastmatrizen \mathbf{C}_e :

$$\mathbf{C}_{e1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix} \quad \mathbf{C}_{e2} = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad \mathbf{C}_{e1 \times 2} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \\ -1 & 0 \\ 0 & -1 \\ 1 & 1 \end{pmatrix}$$

Die ursprünglichen Parameter $\beta_0, \beta_{1,jk}^*, \beta_{2,jk}^*$ und $\beta_{1 \times 2,jk}^*$ erhalten durch die Nebenbedingungen die Bedeutung der Parameter μ, α_j, β_k und $(\alpha\beta)_{jk}$ der häufig gewählten Formulierung des

Modells der zweifaktoriellen Varianzanalyse als $\mu_{jk} = \mu + \alpha_j + \beta_k + (\alpha\beta)_{jk}$.⁵⁷ Mit Effektcodierung lautet das Modell mit reduzierter Designmatrix \mathbf{X} und dem reduzierten Parametervektor $\boldsymbol{\beta}$:

$$\begin{aligned} E(\mathbf{y}) &= \begin{pmatrix} \mu_{11} \\ \mu_{21} \\ \mu_{31} \\ \mu_{12} \\ \mu_{22} \\ \mu_{32} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & -1 & -1 & 1 & -1 & -1 \\ 1 & 1 & 0 & -1 & -1 & 0 \\ 1 & 0 & 1 & -1 & 0 & -1 \\ 1 & -1 & -1 & -1 & 1 & 1 \end{pmatrix}_{\mathbf{1} \quad \mathbf{X}_1 \quad \mathbf{X}_2 \quad \mathbf{X}_{1 \times 2}} \mathbf{x} \begin{pmatrix} \beta_0 \\ \beta_{1,1} \\ \beta_{1,2} \\ \beta_{2,1} \\ \beta_{1 \times 2,11} \\ \beta_{1 \times 2,21} \end{pmatrix}_{\boldsymbol{\beta}} \\ &= \begin{pmatrix} \beta_0 + \beta_{1,1} + \beta_{2,1} + \beta_{1 \times 2,11} \\ \beta_0 + \beta_{1,2} + \beta_{2,1} + \beta_{1 \times 2,21} \\ \beta_0 - (\beta_{1,1} + \beta_{1,2}) + \beta_{2,1} - (\beta_{1 \times 2,11} + \beta_{1 \times 2,21}) \\ \beta_0 + \beta_{1,1} - \beta_{2,1} - \beta_{1 \times 2,11} \\ \beta_0 + \beta_{1,2} - \beta_{2,1} - \beta_{1 \times 2,21} \\ \beta_0 - (\beta_{1,1} + \beta_{1,2}) - \beta_{2,1} + \beta_{1 \times 2,11} + \beta_{1 \times 2,21} \end{pmatrix} \end{aligned}$$

12.9.4 Parameterschätzungen, Vorhersage und Residuen

Der n -Vektor der Beobachtungen $\mathbf{y} = (y_1, \dots, y_n)^t$ ergibt sich im Modell des ALM als Summe von $E(\mathbf{y})$ und einem n -Vektor zufälliger Fehler $\boldsymbol{\epsilon} = (\epsilon_1, \dots, \epsilon_n)^t$.

$$\begin{aligned} y_i &= E(y_i) + \epsilon_i \\ \mathbf{y} &= \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \end{aligned}$$

Die Fehler sollen dabei gemeinsam unabhängig und auch unabhängig von \mathbf{X} sein. Ferner soll für alle $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ gelten, womit $\boldsymbol{\epsilon}$ multivariat normalverteilt ist mit $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})$. Die Voraussetzung gleicher Fehlervarianzen wird im Fall der Regression als Homoskedastizität bezeichnet, im Fall der Varianzanalyse als Varianzhomogenität (Abb. 12.5). Die Beobachtungen sind dann ihrerseits multivariat normalverteilt mit $\mathbf{y} \sim \mathcal{N}(\mathbf{X}\boldsymbol{\beta}, \sigma^2 \mathbf{I})$.⁵⁸

Der Vektor $\hat{\boldsymbol{\beta}}$ der i. S. der geringsten Quadratsumme der Residuen optimalen Parameterschätzungen ist gleich $\mathbf{X}^+ \mathbf{y} = (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{X}^t \mathbf{y}$.⁵⁹ Man erhält $\hat{\boldsymbol{\beta}}$ also als Koordinaten des Kriteriums,

⁵⁷Hierbei seien $\alpha_j = \mu_j - \mu$ die Effektgrößen des Haupteffekts der ersten UV, $\beta_k = \mu_k - \mu$ die des Haupteffekts der zweiten UV und $(\alpha\beta)_{jk} = \mu_{jk} - (\mu + \alpha_j + \beta_k) = \mu_{jk} - \mu_j - \mu_k + \mu$ die der Interaktion. Dabei seien $\mu_j = \frac{1}{q} \sum_k \mu_{jk}$, $\mu_k = \frac{1}{p} \sum_j \mu_{jk}$ und $\mu = \frac{1}{p \cdot q} \sum_j \sum_k \mu_{jk}$ ungewichtete mittlere Erwartungswerte. Liegen gleiche, oder zumindest proportional ungleiche Zellbesetzungen vor ($\frac{n_{jk}}{n_{j'k'}} = \frac{n_{j'k}}{n_{j'k'}}$ sowie $\frac{n_{jk}}{n_{j'k}} = \frac{n_{jk'}}{n_{j'k'}}$ für alle j, j', k, k'), lässt sich die Parametrisierung mit gewichteten mittleren Erwartungswerten durch die gewichtete Effektcodierung umsetzen (vgl. Fußnote 55).

⁵⁸Zunächst ist $E(\mathbf{y}) = E(\mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}) = \mathbf{X}\boldsymbol{\beta} + E(\boldsymbol{\epsilon}) = \mathbf{X}\boldsymbol{\beta}$. Weiter gilt $V(\mathbf{y}) = V(\mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}) = V(\boldsymbol{\epsilon}) = \sigma^2 \mathbf{I}$.

⁵⁹In der Varianzanalyse werden die reduzierten Parameter (Kontraste) geschätzt. Für die Beziehung zwischen geschätzten ursprünglichen Parametern und geschätzten Kontrasten gilt $\hat{\boldsymbol{\beta}}^* = C\hat{\boldsymbol{\beta}}$. Auf Basis eines mit `aov()` oder `lm()` angepassten linearen Modells erhält man $\hat{\boldsymbol{\beta}}$ mit `coef()` und $\hat{\boldsymbol{\beta}}^*$ im univariaten Fall mit `dummy.coef()`.

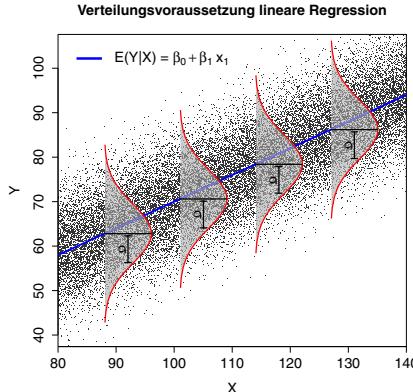


Abbildung 12.5: Verteilungsvoraussetzungen im allgemeinen linearen Modell am Beispiel der einfachen linearen Regression

das orthogonal auf V projiziert wurde, bzgl. der durch \mathbf{X} definierten Basis (vgl. Abschn. 12.1.7). Es gilt $\hat{\beta} \sim \mathcal{N}(\beta, \sigma^2(\mathbf{X}^t \mathbf{X})^{-1})$, $\hat{\beta}$ ist damit erwartungstreuer Schätzer für β .⁶⁰

Der n -Vektor $\hat{\mathbf{y}} = (\hat{y}_1, \dots, \hat{y}_n)^t$ der vorhergesagten Werte berechnet sich durch $\mathbf{X} \mathbf{X}^+ \mathbf{y} = \mathbf{P} \mathbf{y}$, wobei die orthogonale Projektion $\mathbf{P} = \mathbf{X}(\mathbf{X}^t \mathbf{X})^{-1} \mathbf{X}^t$ auch als *Hat-Matrix* bezeichnet wird. Man erhält $\hat{\mathbf{y}}$ also als Koordinaten des Kriteriums, das orthogonal auf V projiziert wurde, bzgl. der Standardbasis. Die Vorhersage $\hat{\mathbf{y}}$ liegt damit in V .

Für den n -Vektor der Residuen $\mathbf{e} = (e_1, \dots, e_n)^t$ gilt $\mathbf{e} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{I} \mathbf{y} - \mathbf{P} \mathbf{y} = (\mathbf{I} - \mathbf{P}) \mathbf{y}$. Die Residuen ergeben sich also als Projektion des Kriteriums auf das orthogonale Komplement von V . Die Residuen \mathbf{e} liegen damit in V^\perp und sind senkrecht zur Vorhersage. V^\perp wird auch als *Fehlerraum* bezeichnet und besitzt die Dimension $\text{Rang}(\mathbf{I} - \mathbf{P}) = n - \text{Rang}(\mathbf{X})$. Weiter gilt $E(\mathbf{e}) = \mathbf{0}$ und $V(\mathbf{e}) = \sigma^2(\mathbf{I} - \mathbf{P})$.⁶¹

Erwartungstreuer Schätzer für σ^2 ist die Quadratsumme der Residuen $SS_e = \|\mathbf{e}\|^2 = \mathbf{y}^t (\mathbf{I} - \mathbf{P}) \mathbf{y}$ geteilt durch die Anzahl der Fehler-Freiheitsgrade, die gleich der Dimension von V^\perp ist.⁶² In der Regression mit p Prädiktoren und absolutem Term ist $\hat{\sigma}^2 = \frac{\|\mathbf{e}\|^2}{n-(p+1)}$ der quadrierte Standardschätzfehler, in der einfaktoriellen Varianzanalyse mit p Gruppen ist $\hat{\sigma}^2 = \frac{\|\mathbf{e}\|^2}{n-p}$ die mittlere Quadratsumme der Residuen.

Im multivariaten Fall werden mehrere Kriteriums- bzw. AV-Vektoren \mathbf{y}_l spaltenweise zu einer Matrix \mathbf{Y} zusammengestellt. Setzt man in den genannten Formeln \mathbf{Y} für \mathbf{y} ein, berechnen sich

⁶⁰Zunächst ist $E(\hat{\beta}) = E(\mathbf{X}^+ \mathbf{y}) = \mathbf{X}^+ E(\mathbf{y}) = (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{X}^t \mathbf{X} \beta = \beta$. Weiter gilt $V(\hat{\beta}) = V(\mathbf{X}^+ \mathbf{y}) = \mathbf{X}^+ V(\mathbf{y}) \mathbf{X}^{+t} = (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{X}^t \sigma^2 \mathbf{I} \mathbf{X} (\mathbf{X}^t \mathbf{X})^{-1} = \sigma^2 (\mathbf{X}^t \mathbf{X})^{-1}$.

⁶¹Zunächst ist $E(\mathbf{e}) = E((\mathbf{I} - \mathbf{P}) \mathbf{y}) = E(\mathbf{y}) - E(\mathbf{P} \mathbf{y}) = \mathbf{X} \beta - \mathbf{P} \mathbf{X} \beta$. Da die Spalten von \mathbf{X} in V liegen, bleiben sie durch \mathbf{P} unverändert, es folgt also $E(\mathbf{e}) = \mathbf{X} \beta - \mathbf{P} \mathbf{X} \beta = \mathbf{0}$. Weiter gilt $V(\mathbf{e}) = V((\mathbf{I} - \mathbf{P}) \mathbf{y}) = (\mathbf{I} - \mathbf{P}) V(\mathbf{y}) (\mathbf{I} - \mathbf{P})^t = \sigma^2 (\mathbf{I} - \mathbf{P}) (\mathbf{I} - \mathbf{P})^t$. Als orthogonale Projektion ist $\mathbf{I} - \mathbf{P}$ symmetrisch und idempotent, es gilt also $V(\mathbf{e}) = \sigma^2 (\mathbf{I} - \mathbf{P}) (\mathbf{I} - \mathbf{P})^t = \sigma^2 (\mathbf{I} - \mathbf{P})$.

⁶² $SS_e = \sum_i e_i^2 = \|\mathbf{e}\|^2 = \mathbf{e}^t \mathbf{e} = (\mathbf{y} - \hat{\mathbf{y}})^t (\mathbf{y} - \hat{\mathbf{y}}) = \mathbf{y}^t \mathbf{y} - \mathbf{y}^t \hat{\mathbf{y}} - \hat{\mathbf{y}}^t \mathbf{y} + \hat{\mathbf{y}}^t \hat{\mathbf{y}} = \mathbf{y}^t \mathbf{y} - \mathbf{y}^t \mathbf{P} \mathbf{y} - (\mathbf{P} \mathbf{y})^t \mathbf{y} + (\mathbf{P} \mathbf{y})^t (\mathbf{P} \mathbf{y}) = \mathbf{y}^t \mathbf{y} - \mathbf{y}^t \mathbf{P} \mathbf{y} - \mathbf{y}^t \mathbf{P} \mathbf{P} \mathbf{y} + \mathbf{y}^t \mathbf{P} \mathbf{P} \mathbf{y} = \mathbf{y}^t \mathbf{y} - \mathbf{y}^t \mathbf{P} \mathbf{y} = \mathbf{y}^t (\mathbf{I} - \mathbf{P}) \mathbf{y}$.

Parameterschätzungen, die spaltenweise aus den Vektoren der Vorhersagen $\hat{\mathbf{y}}_l$ zusammengestellte Matrix $\hat{\mathbf{Y}}$ und die spaltenweise aus den Vektoren der Residuen \mathbf{e}_l zusammengestellte Matrix \mathbf{E} wie in der univariaten Formulierung.

12.9.5 Hypothesen über parametrische Funktionen testen

Im ALM können verschiedenartige Hypothesen über die Modellparameter β_j getestet werden. Eine Gruppe solcher Hypothesen ist jene über den Wert einer *parametrischen Funktion*. Im univariaten Fall ist eine parametrische Funktion $\psi = \sum_j c_j \beta_j = \mathbf{c}^t \boldsymbol{\beta}$ eine Linearkombination der Parameter, wobei die Koeffizienten c_j zu einem Vektor \mathbf{c} zusammengestellt werden. Beispiele für parametrische Funktionen sind etwa a-priori Kontraste aus der Varianzanalyse (vgl. Abschn. 7.3.6). Auch ein Parameter β_j selbst (z. B. ein Gewicht in der Regression) ist eine parametrische Funktion, bei der \mathbf{c} der j -te Einheitsvektor ist. Die H_0 lässt sich dann als $\psi = \psi_0$ mit einem festen Wert ψ_0 formulieren.

Eine parametrische Funktion wird mit Hilfe der Parameterschätzungen als $\hat{\psi} = \mathbf{c}^t \hat{\boldsymbol{\beta}}$ erwartungstreugeschätzt, und es gilt $\hat{\psi} \sim \mathcal{N}(\psi, \sigma^2 \mathbf{c}^t (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{c})$.⁶³ $\hat{\psi}$ lässt sich auch direkt als Linearkombination der Beobachtungen y_i im Vektor \mathbf{y} formulieren:

$$\hat{\psi} = \mathbf{c}^t \hat{\boldsymbol{\beta}} = \mathbf{c}^t (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{X}^t \mathbf{y} = (\mathbf{X} (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{c})^t \mathbf{y} = \mathbf{a}^t \mathbf{y}$$

Dabei ist $\mathbf{a} = \mathbf{X} (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{c}$ der zu \mathbf{c} gehörende n -Vektor der *Schätzerkoeffizienten*, mit dem auch $\hat{\psi} = \mathbf{a}^t \hat{\mathbf{y}}$ ⁶⁴ sowie $\sigma_{\hat{\psi}}^2 = \mathbf{a}^t \mathbf{a} \sigma^2 = \|\mathbf{a}\|^2 \sigma^2$ gilt.⁶⁵ Damit lässt sich die t -Teststatistik in der üblichen Form $\frac{\hat{\psi} - \psi_0}{\hat{\sigma}_{\hat{\psi}}}$ definieren:

$$t = \frac{\hat{\psi} - \psi_0}{\hat{\sigma} \sqrt{\mathbf{c}^t (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{c}}} = \frac{\hat{\psi} - \psi_0}{\|\mathbf{a}\| \sqrt{\|\mathbf{e}\|^2 / (n - \text{Rang}(\mathbf{X}))}}$$

Unter H_0 ist t zentral t -verteilt mit $n - \text{Rang}(\mathbf{X})$ Freiheitsgraden (im Fall der Regression $n - (p + 1)$, im Fall der einfaktoriellen Varianzanalyse $(n - p)$).

12.9.6 Lineare Hypothesen als Modellvergleiche formulieren

Analog zur einzelnen parametrischen Funktion können Hypothesen über einen Vektor ψ von m parametrischen Funktionen $\psi_j = \mathbf{c}_j^t \boldsymbol{\beta}$ gleichzeitig formuliert werden, wobei die ψ_j linear unabhängig sein sollen. Diese *linearen* Hypothesen besitzen die Form $\psi = \mathbf{L} \boldsymbol{\beta}$. Dabei ist \mathbf{L} eine Matrix aus den zeilenweise zusammengestellten Koeffizientenvektoren \mathbf{c}_j für die Linearkombinationen der allgemein u Parameter im Vektor $\boldsymbol{\beta}$ ($u > m$). \mathbf{L} ist dann eine $(m \times u)$ -Matrix mit Rang m . Der unter H_0 für ψ angenommene Vektor sei ψ_0 .

⁶³Zunächst ist $E(\hat{\psi}) = E(\mathbf{c}^t \hat{\boldsymbol{\beta}}) = \mathbf{c}^t E(\hat{\boldsymbol{\beta}}) = \mathbf{c}^t \boldsymbol{\beta} = \psi$. Weiter gilt $V(\hat{\psi}) = V(\mathbf{c}^t \hat{\boldsymbol{\beta}}) = \mathbf{c}^t V(\hat{\boldsymbol{\beta}}) \mathbf{c} = \mathbf{c}^t \sigma^2 (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{c}$. Bei $\hat{\psi}$ handelt es sich um einen *Gauß-Markoff-Schätzer*, also den linearen erwartungstreuen Schätzer mit der geringsten Varianz.

⁶⁴Zunächst ist $\mathbf{X}^t \mathbf{a} = \mathbf{X}^t \mathbf{X} (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{c} = \mathbf{c}$, also $\mathbf{c}^t = \mathbf{a}^t \mathbf{X}$. Damit gilt $\mathbf{a}^t \mathbf{y} = \mathbf{c}^t (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{X}^t \mathbf{y} = \mathbf{a}^t \mathbf{X} (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{X}^t \mathbf{y} = \mathbf{a}^t \mathbf{P} \mathbf{y} = \mathbf{a}^t \hat{\mathbf{y}}$.

⁶⁵ $\|\mathbf{a}\|^2 = \mathbf{a}^t \mathbf{a} = (\mathbf{X} (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{c})^t \mathbf{X} (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{c} = \mathbf{c}^t (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{X}^t \mathbf{X} (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{c} = \mathbf{c}^t (\mathbf{X}^t \mathbf{X})^{-1} \mathbf{c}$.

Im multivariaten Fall mit r Variablen Y_i und der $(u \times r)$ -Matrix der Parameter \boldsymbol{B} hat eine lineare Hypothese die Form $\boldsymbol{\Psi} = \mathbf{L}\boldsymbol{B}$, wobei die unter H_0 für $\boldsymbol{\Psi}$ angenommene Matrix $\boldsymbol{\Psi}_0$ sei.

Die hier vorgestellten linearen Hypothesen aus der Varianzanalyse und Regression haben die Form $\boldsymbol{\psi}_0 = \mathbf{0}$ (bzw. $\boldsymbol{\Psi}_0 = \mathbf{0}$) und beziehen sich auf den Vergleich zweier nested Modelle (vgl. Abschn. 6.3.3, 7.3.3): Das umfassendere (*unrestricted*) H_1 -Modell mit Designmatrix \mathbf{X}_u besitzt dabei im univariaten Fall allgemein u freie Parameter. Für das eingeschränkte (*restricted*) H_0 -Modell mit Designmatrix \mathbf{X}_r , nimmt man an, dass m der Parameter des umfassenderen Modells 0 sind, wodurch es noch $r = u - m$ freie Parameter besitzt. Man erhält \mathbf{X}_r , indem man die zu den auf 0 gesetzten Parametern gehörenden m Spalten von \mathbf{X}_u streicht.

Die freien Parameter des eingeschränkten Modells bilden eine echte Teilmenge jener des umfassenderen Modells. Daher liegt der von den Spalten von \mathbf{X}_r aufgespannte Unterraum V_r der Vorhersage des eingeschränkten Modells vollständig in jenem des umfassenderen Modells V_u , dem Erzeugnis der Spalten von \mathbf{X}_u . Umgekehrt liegt der Fehlerraum des umfassenderen Modells V_u^\perp vollständig in jenem des eingeschränkten Modells V_r^\perp . Die H_0 lässt sich auch so formulieren, dass $E(\mathbf{y})$ in V_r liegt.

Im multivariaten Fall besitzt \boldsymbol{B} allgemein u Zeilen mit freien Parametern. Für das eingeschränkte H_0 -Modell nimmt man an, dass m Zeilen von \boldsymbol{B} gleich 0 sind. Dadurch besitzt \boldsymbol{B} unter H_0 noch $r = u - m$ Zeilen mit freien Parametern. \mathbf{X}_u und \mathbf{X}_r stehen wie im univariaten Fall zueinander, unter H_0 soll also jede Spalte von $E(\mathbf{Y})$ in V_r liegen. Für die inferenzstatistische Prüfung vgl. Abschn. 12.9.7.

Univariate einfaktorielle Varianzanalyse

In der univariaten einfaktoriellen Varianzanalyse sind unter H_0 alle Gruppenerwartungswerte gleich. Dies ist äquivalent zur Hypothese, dass $\beta_{p-1} = \mathbf{0}$, also $\boldsymbol{\beta} = (\beta_0, \beta_{p-1}^t)^t = (\beta_0, \mathbf{0}^t)^t$ gilt. In $H_0 : \mathbf{C}\boldsymbol{\beta} = \mathbf{0}$ hat \mathbf{C} hier damit die Form $[\mathbf{0} | \mathbf{I}]$, wobei $\mathbf{0}$ der $(p-1)$ -Vektor $(0, \dots, 0)^t$ und \mathbf{I} die $((p-1) \times (p-1))$ -Einheitsmatrix ist. Das H_0 -Modell mit einem Parameter β_0 lässt sich als $E(\mathbf{y}) = \mathbf{1}\beta_0$ formulieren. Im umfassenderen H_1 -Modell mit p Parametern gibt es keine Einschränkung für β_{p-1} , es ist damit identisch zum vollständigen Modell der einfaktoriellen Varianzanalyse $E(\mathbf{y}) = \mathbf{X}\boldsymbol{\beta}$: Hier ist also $u = p$, $\mathbf{X}_u = \mathbf{X}$, $r = 1$, $\mathbf{X}_r = \mathbf{1}$ und $m = p - 1$ (vgl. Abschn. 12.9.2).

Univariate zweifaktorielle Varianzanalyse: Quadratsummen vom Typ I

Wie in Abschn. 7.5.2 erläutert, existieren in der zweifaktoriellen Varianzanalyse (CRF- pq Design) Typen von Quadratsummen, die bei ungleichen Zellbesetzungen zu unterschiedlichen Tests führen können. Proportional ungleiche Zellbesetzungen sind dabei gegeben, wenn $\frac{n_{jk}}{n_{j'k'}} = \frac{n_{j'k}}{n_{j'k'}}$ sowie $\frac{n_{jk}}{n_{j'k}} = \frac{n_{jk'}}{n_{j'k'}}$ für alle j, j', k, k' gilt. Ist dies nicht der Fall, spricht man von unbalancierten Zellbesetzungen.

Zunächst sind für den Test jedes Effekts (beide Haupteffekte und Interaktionseffekt) das eingeschränkte H_0 -Modell sowie das umfassendere H_1 -Modell zu definieren. Die H_0 des Haupteffekts der ersten UV lässt sich mit Hilfe des zugehörigen Parametervektors als $\boldsymbol{\beta}_1 = \mathbf{0}$ formulieren. Bei

Quadratsummen vom Typ I wählt man für den Test des Haupteffekts der UV 1 als H_0 -Modell das Gesamt-Nullmodell, bei dem alle Parametervektoren gleich $\mathbf{0}$ sind und damit $E(\mathbf{y}) = \mathbf{1}\beta_0$ gilt. Hier gilt also für die Designmatrix des eingeschränkten Modells $\mathbf{X}_r = \mathbf{X}_0 = \mathbf{1}$. Das sequentiell folgende H_1 -Modell für den Test der ersten UV ist jenes, bei dem man die zugehörigen $p - 1$ Parameter in β_1 dem eingeschränkten Modell hinzufügt und die Designmatrix entsprechend erweitert, d. h. $\mathbf{X}_u = [\mathbf{1}|\mathbf{X}_1]$:

$$E(\mathbf{y}) = [\mathbf{1}|\mathbf{X}_1] \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix}$$

Die H_0 des Haupteffekts der UV 2 lautet mit Hilfe des zugehörigen Parametervektors $\beta_2 = \mathbf{0}$. Bei Quadratsummen vom Typ I fällt die Wahl für das H_0 -Modell der zweiten UV auf das H_1 -Modell der ersten UV ($\mathbf{X}_r = [\mathbf{1}|\mathbf{X}_1]$). Das H_1 -Modell für den Test der zweiten UV ist jenes mit der sequentiell erweiterten Designmatrix $\mathbf{X}_u = [\mathbf{1}|\mathbf{X}_1|\mathbf{X}_2]$:

$$E(\mathbf{y}) = [\mathbf{1}|\mathbf{X}_1|\mathbf{X}_2] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}$$

Durch den sequentiellen Aufbau der Modellvergleiche ist die Reihenfolge der UVn beim Test der Haupteffekte in Fällen mit unbalancierten Zellbesetzungen rechnerisch bedeutsam.

Die H_0 des Interaktionseffekts lautet mit Hilfe des zugehörigen Parametervektors $\beta_{1 \times 2} = \mathbf{0}$. Die Wahl für das H_0 -Modell der Interaktion ist das sequentielle H_1 -Modell der UV 2 ($\mathbf{X}_r = [\mathbf{1}|\mathbf{X}_1|\mathbf{X}_2]$). Das H_1 -Modell der Interaktion ist das in Abschn. 12.9.3 vorgestellte vollständige Modell der zweifaktoriellen Varianzanalyse ($\mathbf{X}_u = \mathbf{X}$):

$$E(\mathbf{y}) = [\mathbf{1}|\mathbf{X}_1|\mathbf{X}_2|\mathbf{X}_{1 \times 2}] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_{1 \times 2} \end{pmatrix} = \mathbf{X}\beta$$

In $H_0 : \mathbf{C}\beta = \mathbf{0}$ hat \mathbf{C} in den genannten Tests der Haupteffekte die Form $[\mathbf{0}|\mathbf{I}|\mathbf{0}]$ und beim Test der Interaktion die Form $[\mathbf{0}|\mathbf{I}]$. Dabei ist \mathbf{I} der Reihe nach die $((p - 1) \times (p - 1))$ -, $((q - 1) \times (q - 1))$ - und $((p - 1) \cdot (q - 1) \times (p - 1) \cdot (q - 1))$ -Einheitsmatrix und $\mathbf{0}$ eine passende Matrix aus 0-Einträgen.

Zweifaktorielle Varianzanalyse: Quadratsummen vom Typ II

Auch bei Quadratsummen vom Typ II unterscheiden sich die eingeschränkten und umfassenderen Modelle um jeweils $p - 1$ (erster Haupteffekt), $q - 1$ (zweiter Haupteffekt) und $(p - 1) \cdot (q - 1)$ (Interaktion) freie Parameter. Das H_1 -Modell beim Test beider Haupteffekte ist hier:

$$E(\mathbf{y}) = [\mathbf{1}|\mathbf{X}_1|\mathbf{X}_2] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}$$

Das jeweilige H_0 -Modell für den Test des Haupteffekts einer UV wird bei Quadratsummen vom Typ II dadurch gebildet, dass ausgehend vom H_1 -Modell die Parameter des zu testenden Effekts auf 0 gesetzt und die zugehörigen Spalten der Designmatrix entfernt werden.

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_2] \begin{pmatrix} \beta_0 \\ \beta_2 \end{pmatrix} \quad (H_0 - \text{Modell für Test der UV 1})$$

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_1] \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} \quad (H_0 - \text{Modell für Test der UV 2})$$

Die Reihenfolge der UVn ist anders als bei Quadratsummen vom Typ I beim Test der Haupteffekte unwesentlich, selbst wenn unbalancierte Zellbesetzungen vorliegen. Liegen proportional ungleiche Zellbesetzungen vor, stimmen die Ergebnisse für den Test der Haupteffekte mit jenen aus Quadratsummen vom Typ I überein. Bei Quadratsummen vom Typ II ist die Wahl für H_0 - und H_1 -Modell beim Test der Interaktion gleich jener bei Quadratsummen vom Typ I und III, die Ergebnisse sind daher identisch.

Zweifaktorielle Varianzanalyse: Quadratsummen vom Typ III

Auch bei Quadratsummen vom Typ III unterscheiden sich die eingeschränkten und umfassenderen Modelle um jeweils $p - 1$ (erster Haupteffekt), $q - 1$ (zweiter Haupteffekt) und $(p - 1) \cdot (q - 1)$ (Interaktion) freie Parameter. Hier ist beim Test aller Effekte das H_1 -Modell das in Abschn. 12.9.3 vorgestellte vollständige Modell der zweifaktoriellen Varianzanalyse:

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_1 | \mathbf{X}_2 | \mathbf{X}_{1 \times 2}] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_{1 \times 2} \end{pmatrix} = \mathbf{X}\boldsymbol{\beta}$$

Das jeweilige H_0 -Modell für den Test aller Effekte wird bei Quadratsummen vom Typ III dadurch gebildet, dass ausgehend vom vollständigen Modell die Parameter des zu testenden Effekts auf 0 gesetzt und die zugehörigen Spalten der Designmatrix gestrichen werden:

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_2 | \mathbf{X}_{1 \times 2}] \begin{pmatrix} \beta_0 \\ \beta_2 \\ \beta_{1 \times 2} \end{pmatrix} \quad (H_0 - \text{Modell für Test UV 1})$$

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_1 | \mathbf{X}_{1 \times 2}] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_{1 \times 2} \end{pmatrix} \quad (H_0 - \text{Modell für Test UV 2})$$

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_1 | \mathbf{X}_2] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix} \quad (H_0 - \text{Modell für Test Interaktion})$$

Die Reihenfolge der UVn ist anders als bei Quadratsummen vom Typ I beim Test der Haupteffekte unwesentlich, selbst wenn unbalancierte Zellbesetzungen vorliegen. Bei gleichen Zellbesetzungen liefern alle drei Typen von Quadratsummen dieselben Ergebnisse. Bei Quadratsummen vom Typ III stimmt die Wahl für H_0 - und H_1 -Modell beim Test der Interaktion mit jener bei Quadratsummen vom Typ I und II überein, die Ergebnisse sind daher identisch. Ein Test des absoluten Terms β_0 ist mit Quadratsummen vom Typ III ebenso wenig möglich wie ein Test von Haupteffekten, die nicht Teil einer Interaktion sind.

Die H_0 -Modelle für Haupteffekte machen bei Quadratsummen vom Typ III keine Einschränkung für den Parametervektor $\beta_{1 \times 2}$ der Interaktion, obwohl die Parameter des zu testenden, und an der Interaktion beteiligten Haupteffekts auf 0 gesetzt werden. Die Modellvergleiche verletzen damit anders als bei Quadratsummen vom Typ I und II das Prinzip, dass Vorhersageterme höherer Ordnung nur in das Modell aufgenommen werden, wenn alle zugehörigen Terme niederer Ordnung ebenfalls Berücksichtigung finden.

Modellvergleiche für Quadratsummen vom Typ III können deshalb bei ungleichen Zellbesetzungen in Abhängigkeit von der verwendeten Nebenbedingung für die Parameter samt Codierschema (vgl. Abschn. 12.9.2) unterschiedlich ausfallen. Richtige Ergebnisse erhält man mit $v = 1$, was alle Codierschemata gewährleisten, deren Koeffizienten sich über die Spalten von C zu 0 summieren. Dazu zählen Effekt- und Helmert-Codierung, nicht aber Treatment-Kontraste (Dummy-Codierung) – die Voreinstellung in R. Bei Quadratsummen vom Typ I und II spielt die Wahl von Nebenbedingung und Codierschema dagegen keine Rolle für den inferenzstatistischen Test.

Multiple Regression

Die Modellvergleiche beim Test von p Prädiktoren X_j in der multiplen Regression werden nach demselben Prinzip wie in der zweifaktoriellen Varianzanalyse gebildet: Das Gesamt-Nullmodell ist $E(\mathbf{Y}) = \mathbf{1}\beta_0^t$, bei dem für alle Kriterien \mathbf{y}_l der zugehörige Parametervektor der p Prädiktoren $\beta_{p,l} = \mathbf{0}$ und damit $\mathbf{B}_{p,0} = \mathbf{0}$ sowie $\mathbf{X}_0 = \mathbf{1}$ ist. Das vollständige Modell ist jenes mit allen Prädiktoren $E(\mathbf{Y}) = \mathbf{XB}$.

Die Prädiktoren werden über Modellvergleiche getestet, die wie in der zweifaktoriellen Varianzanalyse vom verwendeten Typ der Quadratsummen abhängen. Die Reihenfolge der Prädiktoren ist im multivariaten Fall bei Quadratsummen vom Typ I bedeutsam, sofern nicht alle Prädiktoren paarweise unkorreliert sind: Für den Test des ersten Prädiktors ist das eingeschränkte H_0 -Modell das Gesamt-Nullmodell, das zugehörige umfassendere H_1 -Modell ist jenes mit nur dem ersten Prädiktor. Für den Test des zweiten Prädiktors ist das eingeschränkte H_0 -Modell das sequentielle H_1 -Modell des ersten Prädiktors, das H_1 -Modell ist jenes mit den ersten beiden Prädiktoren. Für die Tests aller folgenden Prädiktoren gilt dies analog.

Bei Quadratsummen vom Typ II ist das H_1 -Modell beim Test aller Prädiktoren gleich dem vollständigen Modell mit allen Prädiktoren, jedoch ohne deren Interaktionen. Das H_0 -Modell beim Test eines Prädiktors X_j entsteht aus dem H_1 -Modell, indem der zu X_j gehörende Parametervektor $\beta_j^t = \mathbf{0}^t$ gesetzt und die passende Spalte der Designmatrix gestrichen wird.

Bei Quadratsummen vom Typ III ist das H_1 -Modell beim Test aller Prädiktoren gleich dem vollständigen Modell mit allen Prädiktoren und deren Interaktionen, sofern letztere berücksichtigt werden sollen. Das H_0 -Modell beim Test eines Prädiktors X_j entsteht aus dem H_1 -Modell, indem der zu X_j gehörende Parametervektor $\beta_j^t = \mathbf{0}^t$ gesetzt und die passende Spalte der Designmatrix gestrichen wird. Quadratsummen vom Typ II und III unterscheiden sich also nur, wenn die Regression auch Interaktionsterme einbezieht.

Sollen allgemein die Parameter von m Prädiktoren gleichzeitig daraufhin getestet werden, ob sie 0 sind, ist das Vorgehen analog, wobei unter H_0 die zugehörigen, aus \mathbf{B} stammenden m Parametervektoren $\beta_j^t = \mathbf{0}^t$ gesetzt und entsprechend ausgehend vom H_1 -Modell die passenden Spalten der Designmatrix gestrichen werden.

12.9.7 Lineare Hypothesen testen

Liegt eine lineare Hypothese der Form $\psi = \mathbf{L}\beta$ vor, soll \mathbf{A} die zur $((u-r) \times u)$ -Matrix \mathbf{L} gehörende $((u-r) \times n)$ -Matrix bezeichnen, die zeilenweise aus den n -Vektoren der Schätzerkoeffizienten $a_j = \mathbf{X}_u(\mathbf{X}_u^t \mathbf{X}_u)^{-1} c_j$ gebildet wird (vgl. Abschn. 12.9.5). Es gilt also $\mathbf{A}^t = \mathbf{X}_u(\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{L}^t$ und $\text{Rang}(\mathbf{A}) = \text{Rang}(\mathbf{L}) = u - r$. Aus $c = \mathbf{X}^t \alpha$ (vgl. Abschn. 12.9.6, Fußnote 64) folgt damit $\mathbf{L}^t = \mathbf{X}_u^t \mathbf{A}^t$, also $\mathbf{L} = \mathbf{A} \mathbf{X}_u$.⁶⁶

Der Vektor der Schätzungen $\hat{\psi} = \mathbf{L}\hat{\beta}$ kann mit \mathbf{A} analog zur Schätzung einer einfachen parametrischen Funktion $\hat{\psi}_j = \mathbf{c}_j^t \hat{\beta} = \mathbf{a}_j^t \mathbf{y}$ auch als Abbildung des Vektors der Beobachtungen \mathbf{y} formuliert werden:

$$\hat{\psi} = \mathbf{L}\hat{\beta} = \mathbf{L}(\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{X}_u^t \mathbf{y} = \mathbf{A}\mathbf{y}$$

Mit der Vorhersage $\hat{\mathbf{y}}$ gilt zudem $\hat{\psi} = \mathbf{A}\hat{\mathbf{y}}$.⁶⁷ $\hat{\psi}$ ist ein erwartungstreuer Gauß-Markoff-Schätzer mit Verteilung $\hat{\psi} \sim \mathcal{N}(\psi, \sigma^2 \mathbf{A}\mathbf{A}^t)$.⁶⁸ Der Rang der Kovarianzmatrix $\Sigma_{\hat{\psi}} = \sigma^2 \mathbf{A}\mathbf{A}^t$ beträgt $u - r$. Er ist also gleich der Differenz der Anzahl zu schätzender Parameter beider Modelle sowie gleich der Differenz der Dimensionen von V_u und V_r einerseits und von V_r^\perp und V_u^\perp andererseits.

Univariate Teststatistik

Um die Teststatistik für den univariaten Fall zu motivieren, ist zunächst festzustellen, dass für die quadrierte Mahalanobisdistanz (vgl. Abschn. 12.1.4) $\|\hat{\psi} - \psi_0\|_{M, \Sigma_{\hat{\psi}}}^2$ von $\hat{\psi}$ zu $\psi_0 = \mathbf{0}$ bzgl. $\Sigma_{\hat{\psi}}$ gilt (vgl. Fußnoten 67 und 68):

$$\begin{aligned} \|\hat{\psi} - \psi_0\|_{M, \Sigma_{\hat{\psi}}}^2 &= (\hat{\psi} - \mathbf{0})^t \Sigma_{\hat{\psi}}^{-1} (\hat{\psi} - \mathbf{0}) = \frac{\hat{\psi}^t (\mathbf{L}(\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{L}^t)^{-1} \hat{\psi}}{\sigma^2} \\ &= \frac{\hat{\psi}^t (\mathbf{A}\mathbf{A}^t)^{-1} \hat{\psi}}{\sigma^2} = \frac{\mathbf{y}^t \mathbf{A}^t (\mathbf{A}\mathbf{A}^t)^{-1} \mathbf{A}\mathbf{y}}{\sigma^2} \end{aligned}$$

⁶⁶ $\mathbf{A}\mathbf{X}_u = (\mathbf{X}_u(\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{L}^t)^t \mathbf{X}_u = \mathbf{L}(\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{X}_u^t \mathbf{X}_u = \mathbf{L}$.

⁶⁷ $\hat{\psi} = \mathbf{L}\hat{\beta} = \mathbf{L}(\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{X}_u^t \mathbf{y} = \mathbf{A}\mathbf{X}_u(\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{X}_u^t \mathbf{y} = \mathbf{A}\hat{\mathbf{y}}$.

⁶⁸ Zunächst gilt $E(\mathbf{L}\hat{\beta}) = LE(\hat{\beta}) = L\beta = \psi$. Weiter ist $V(\hat{\psi}) = V(L\hat{\beta}) = LV(\hat{\beta})L^t = \sigma^2 \mathbf{L}(\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{X}_u^t \mathbf{X}_u(\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{L}^t = \sigma^2 \mathbf{A}\mathbf{A}^t$. Insbesondere ist also $\mathbf{A}\mathbf{A}^t = \mathbf{L}(\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{L}^t$.

Hier ist $\mathbf{A}^t(\mathbf{AA}^t)^{-1}\mathbf{A}$ die Projektion auf das orthogonale Komplement von V_r in V_u (also $V_r^\perp \cap V_u$), dessen spaltenweise Basis \mathbf{A}^t ist.⁶⁹

Für die Schätzung der Fehlervarianz σ^2 benötigt man das vollständige (*full*) Modell mit Designmatrix \mathbf{X}_f und Projektionsmatrix \mathbf{P}_f (bisher einfach als \mathbf{X} und \mathbf{P} bezeichnet), das alle Parameter beinhaltet. Der zugehörige Vektor der Residuen sei $\mathbf{e}_f = \mathbf{y} - \hat{\mathbf{y}}_f$, die Residual-Quadratsumme also $SS_{ef} = \|\mathbf{e}_f\|^2$ mit Freiheitsgraden $df_{ef} = n - \text{Rang}(\mathbf{X}_f)$, der Dimension des Fehlerraumes V_f^\perp (vgl. Abschn. 12.9.4). Als erwartungstreuer Schätzer $\hat{\sigma}^2$ dient in allen Tests $SS_{ef}/df_{ef} = \mathbf{y}^t(\mathbf{I} - \mathbf{P}_f)\mathbf{y}/(n - \text{Rang}(\mathbf{X}_f))$ (vgl. Abschn. 12.9.5, Fußnote 62) – also auch dann, wenn das umfassendere Modell nicht mit dem vollständigen Modell übereinstimmt. Damit lassen sich lineare Nullhypotesen $\mathbf{L}\beta = \mathbf{0}$ im univariaten Fall mit der folgenden Teststatistik prüfen:

$$\begin{aligned} F &= \frac{\hat{\psi}^t(\mathbf{L}(\mathbf{X}_u^t\mathbf{X}_u)^{-1}\mathbf{L}^t)^{-1}\hat{\psi}/\text{Rang}(\Sigma_{\hat{\psi}})}{\hat{\sigma}^2} = \frac{\hat{\psi}^t(\mathbf{AA}^t)^{-1}\hat{\psi}/(u - r)}{\hat{\sigma}^2} \\ &= \frac{\mathbf{y}^t\mathbf{A}^t(\mathbf{AA}^t)^{-1}\mathbf{Ay}/(\text{Rang}(\mathbf{X}_u) - \text{Rang}(\mathbf{X}_r))}{\mathbf{y}^t(\mathbf{I} - \mathbf{P}_f)\mathbf{y}/(n - \text{Rang}(\mathbf{X}_f))} \end{aligned}$$

Unter H_0 ist F zentral F -verteilt mit $\text{Rang}(\mathbf{X}_u) - \text{Rang}(\mathbf{X}_r)$ Zähler- und $n - \text{Rang}(\mathbf{X}_f)$ Nenner-Freiheitsgraden. Das Modell einer Regression mit p Prädiktoren und absolutem Term hat $n - (p + 1)$ Fehler-Freiheitsgrade, das Modell einer einfaktoriellen Varianzanalyse mit p Gruppen $n - p$.

Der Zähler der Teststatistik lässt sich äquivalent umformulieren, wobei explizit Bezug zum Modellvergleich genommen wird. Dafür sei \mathbf{e}_r der Vektor der Residuen des eingeschränkten Modells mit zugehöriger Projektionsmatrix \mathbf{P}_r und analog \mathbf{e}_u der Vektor der Residuen des umfassenderen Modells mit Projektionsmatrix \mathbf{P}_u . In der einfaktoriellen Varianzanalyse sowie im Gesamt-Test der Regressionsanalyse sind das umfassendere Modell und das vollständige Modell identisch ($\mathbf{P}_u = \mathbf{P}_f$), ebenso ist dort das eingeschränkte Modell gleich dem Gesamt-Nullmodell ($\mathbf{P}_r = \mathbf{P}_0$).

Dann ist die Residual-Quadratsumme des eingeschränkten Modells $SS_{er} = \mathbf{y}^t(\mathbf{I} - \mathbf{P}_r)\mathbf{y}$ mit Freiheitsgraden $df_{er} = n - \text{Rang}(\mathbf{X}_r)$ und die des umfassenderen Modells $SS_{eu} = \mathbf{y}^t(\mathbf{I} - \mathbf{P}_u)\mathbf{y}$ mit Freiheitsgraden $df_{eu} = n - \text{Rang}(\mathbf{X}_u)$. Für die Differenz dieser Quadratsummen gilt $SS_{er} - SS_{eu} = \mathbf{y}^t(\mathbf{P}_u - \mathbf{P}_r)\mathbf{y}$, wobei ihre Dimension $df_{er} - df_{eu} = \text{Rang}(\mathbf{X}_u) - \text{Rang}(\mathbf{X}_r)$ und $\mathbf{P}_u - \mathbf{P}_r = \mathbf{A}^t(\mathbf{AA}^t)^{-1}\mathbf{A}$ ist. Damit lautet die Teststatistik:

$$\begin{aligned} F &= \frac{\mathbf{y}^t(\mathbf{P}_u - \mathbf{P}_r)\mathbf{y}/(\text{Rang}(\mathbf{X}_u) - \text{Rang}(\mathbf{X}_r))}{\mathbf{y}^t(\mathbf{I} - \mathbf{P}_f)\mathbf{y}/(n - \text{Rang}(\mathbf{X}_f))} \\ &= \frac{(SS_{er} - SS_{eu})/(df_{er} - df_{eu})}{SS_{ef}/df_{ef}} \end{aligned}$$

⁶⁹Es sei $\mathbf{y}_r = \mathbf{X}_u \mathbf{r}$ aus V_r und $\mathbf{y}_a = \mathbf{A}^t \mathbf{a}$ aus dem Erzeugnis der Spalten von \mathbf{A}^t mit \mathbf{r} und \mathbf{a} als zugehörigen Koordinatenvektoren bzgl. \mathbf{X}_u und \mathbf{A}^t . Mit $\mathbf{A}^t = \mathbf{X}_u (\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{L}^t$ liegt \mathbf{y}_a als Linearkombination der Spalten von \mathbf{X}_u in V_u . Da \mathbf{y}_r in V_r liegt, gilt $\mathbf{L}\mathbf{y}_r = \mathbf{0}$. Damit folgt $\mathbf{y}_a^t \mathbf{y}_r = (\mathbf{A}^t \mathbf{a})^t \mathbf{X}_u \mathbf{r} = \mathbf{a}^t \mathbf{A} \mathbf{X}_u \mathbf{r} = \mathbf{a}^t \mathbf{L} (\mathbf{X}_u^t \mathbf{X}_u)^{-1} \mathbf{X}_u^t \mathbf{X}_u \mathbf{r} = \mathbf{a}^t \mathbf{L} \mathbf{r} = \mathbf{a}^t \mathbf{0} = 0$, d. h. $\mathbf{y}_a \perp \mathbf{y}_r$. Also liegt \mathbf{y}_a auch in V_r^\perp .

Multivariate Teststatistiken

Die Quadratsummen des univariaten Falls verallgemeinern sich im multivariaten Fall zu folgenden Matrizen, auf denen letztlich die Teststatistiken basieren:

- $\mathbf{T} = \mathbf{Y}^t(\mathbf{I} - \mathbf{P}_0)\mathbf{Y}$: Dies ist gleich der SSP-Matrix der Residuen des Gesamt-Nullmodells und damit gleich der SSP-Matrix der Gesamt-Daten.⁷⁰
- $\mathbf{W} = \mathbf{Y}^t(\mathbf{I} - \mathbf{P}_f)\mathbf{Y}$: Dies ist gleich der SSP-Matrix der Residuen des vollständigen Modells (vgl. Fußnote 70), verallgemeinert also SS_{ef} .
- $\mathbf{B} = \mathbf{Y}^t(\mathbf{P}_u - \mathbf{P}_r)\mathbf{Y}$: Dies ist gleich der SSP-Matrix der Vorhersagedifferenzen beider Modelle,⁷¹ verallgemeinert also $SS_{er} - SS_{eu}$.

In der einfaktoriellen Varianzanalyse handelt es sich bei den Matrizen um multivariate Verallgemeinerungen der Quadratsummen zwischen (between, \mathbf{B}) und innerhalb (within, \mathbf{W}) der Gruppen, sowie der totalen Quadratsumme $\mathbf{T} = \mathbf{B} + \mathbf{W}$. In der Diagonale dieser Gleichung findet sich für jede AV die zugehörige Quadratsummenzerlegung aus der univariaten Varianzanalyse wieder. \mathbf{B} ist hier gleichzeitig die SSP-Matrix der durch die zugehörigen Gruppenzentroide ersetzen Daten.

In der zweifaktoriellen Varianzanalyse ist \mathbf{W} die multivariate Verallgemeinerung der Residual-Quadratsumme. Es gibt nun für den Test der ersten UV eine Matrix \mathbf{B}_1 , für den Test der zweiten UV eine Matrix \mathbf{B}_2 und für den Test der Interaktion eine Matrix $\mathbf{B}_{1 \times 2}$ jeweils als multivariate Verallgemeinerung der zugehörigen Effekt-Quadratsumme. Mit diesen Matrizen gilt bei Quadratsummen vom Typ I dann analog zur univariaten Situation $\mathbf{T} = \mathbf{B}_1 + \mathbf{B}_2 + \mathbf{B}_{1 \times 2} + \mathbf{W}$ (vgl. Abschn. 7.5.2). Bei Quadratsummen vom Typ II und III gilt dies nur in orthogonalen Designs.

In der multivariaten multiplen Regression ist analog für den Test jedes Prädiktors j eine Matrix \mathbf{B}_j zu bilden, die sich aus der Differenzprojektion des zugehörigen Paares von eingeschränktem und umfassenderem Modell ergibt.

Auf Basis der Matrizen \mathbf{B} und \mathbf{W} berechnen sich die üblichen multivariaten Teststatistiken für lineare Hypothesen im ALM, die im univariaten Fall alle äquivalent zum oben aufgeführten F -Bruch sind.

- Wilks' Λ : $\frac{\det(\mathbf{W})}{\det(\mathbf{W} + \mathbf{B})}$
- Roys Maximalwurzel: entweder der größte Eigenwert θ_1 von $(\mathbf{B} + \mathbf{W})^{-1}\mathbf{B}$ oder der größte Eigenwert λ_1 von $\mathbf{W}^{-1}\mathbf{B}$ (so definiert in R). Für die Umrechnung von θ_1 und λ_1 gilt $\lambda_1 = \frac{\theta_1}{1-\theta_1}$ sowie $\theta_1 = \frac{\lambda_1}{1+\lambda_1}$.
- Pillai-Bartlett-Spur: $\text{tr}((\mathbf{B} + \mathbf{W})^{-1}\mathbf{B})$
- Hotelling-Lawley-Spur: $\text{tr}(\mathbf{W}^{-1}\mathbf{B})$

⁷⁰ $\mathbf{X}_0 = \mathbf{1}$, daher ist die Matrix der Residuen $\mathbf{E} = (\mathbf{I} - \mathbf{P}_0)\mathbf{Y} = \mathbf{Q}\mathbf{Y}$ zentriert (vgl. Abschn. 12.1.7, Fußnote 22) und $(\mathbf{Q}\mathbf{Y})^t(\mathbf{Q}\mathbf{Y})$ deren SSP-Matrix. Als orthogonale Projektion ist \mathbf{Q} symmetrisch und idempotent, weshalb $(\mathbf{Q}\mathbf{Y})^t(\mathbf{Q}\mathbf{Y}) = \mathbf{Y}^t\mathbf{Q}^t\mathbf{Q}\mathbf{Y} = \mathbf{Y}^t\mathbf{Q}\mathbf{Y}$ gilt.

⁷¹ Zunächst ist die Matrix der Vorhersagedifferenzen $\hat{\mathbf{Y}}_u - \hat{\mathbf{Y}}_r = \mathbf{P}_u\mathbf{Y} - \mathbf{P}_r\mathbf{Y} = \mathbf{Y} - \mathbf{P}_r\mathbf{Y} - \mathbf{Y} + \mathbf{P}_u\mathbf{Y} = (\mathbf{I} - \mathbf{P}_r)\mathbf{Y} - (\mathbf{I} - \mathbf{P}_u)\mathbf{Y} = \mathbf{E}_r - \mathbf{E}_u$ gleich der Matrix der Differenzen der Residuen. Als Differenz zweier zentrierter Matrizen ist sie damit ihrerseits zentriert. Für die weitere Argumentation vgl. Fußnote 70.

12.9.8 Beispiel: Multivariate multiple Regression

Das Ergebnis der multivariaten multiplen Regression in Abschn. 12.5 lässt sich nun mit dem in Abschn. 12.9.6 und 12.9.7 dargestellten Vorgehen manuell prüfen. Im Beispiel sollen anhand der Prädiktoren Alter, Körpergröße und wöchentliche Dauer sportlicher Aktivitäten die Kriterien Körpergewicht und Gesundheit (i. S. eines geeigneten quantitativen Maßes) vorhergesagt werden. Zunächst sind die Designmatrix \mathbf{X} und die Projektion \mathbf{P}_f für das vollständige Regressionsmodell zu erstellen.⁷²

```
> Y <- cbind(weight, health)                      # Matrix der Kriterien
> X <- cbind(1, height, age, sport)               # Designmatrix
> XR <- model.matrix(~ height + age + sport)    # Designmatrix aus R
> all.equal(X, XR, check.attributes=FALSE)        # Vergleich
[1] TRUE

> Xplus <- solve(t(X) %*% X) %*% t(X)          # X+
> B      <- Xplus %*% Y                          # Parameterschätzungen
> Pf     <- X %*% Xplus                         # Projektion (Hat-Matrix)
> Yhat   <- Pf %*% Y                            # Vorhersage

# Kontrolle: vergleiche manuelle Berechnungen mit R
> fit <- lm(Y ~ height + age + sport)           # mult. Regr.-Modell
> all.equal(B, coef(fit), check.attributes=FALSE)  # Parameter
[1] TRUE

> all.equal(Yhat, fitted(fit), check.attributes=FALSE) # Vorhersage
[1] TRUE
```

Im Gesamt-Nullmodell $E(\mathbf{Y}) = \mathbf{1}\beta_0^t$ der multiplen Regression sind alle Parameter bis auf β_0 gleich 0, Designmatrix \mathbf{X}_0 ist der Vektor $\mathbf{1}$. Es folgt die Berechnung der zugehörigen Projektion \mathbf{P}_0 . Die Residuen ergeben sich aus der Projektion $\mathbf{I} - \mathbf{P}_0$ auf das orthogonale Komplement des von \mathbf{X}_0 aufgespannten Raumes.

```
# Gesamt-Nullmodell
> X0 <- X[, 1, drop=FALSE]                      # Designmatrix X0 = 1-Vektor
> P0 <- X0 %*% solve(t(X0) %*% X0) %*% t(X0)  # Projektion
> Id <- diag(N)                                 # n x n Einheitsmatrix I
> WW <- t(Y) %*% (Id - Pf) %*% Y              # W
```

Schließlich muss für den Test jedes der drei Prädiktoren das zugehörige sequentielle Paar aus eingeschränktem H_0 -Modell und umfassenderem H_1 -Modell mit zugehörigen Designmatrizen \mathbf{X}_r und \mathbf{X}_u sowie ihren orthogonalen Projektionen \mathbf{P}_r und \mathbf{P}_u berechnet werden. Aus $\mathbf{P}_u - \mathbf{P}_r$ ergibt sich für jeden Prädiktor die Matrix \mathbf{B}_j , die in die zugehörigen Teststatistiken eingeht.

```
# Test Prädiktor 1: eingeschränktes Modell (= Gesamt-Nullmodell)
```

⁷²Der gewählte Weg zur Berechnung der Projektionsmatrizen soll die mathematischen Formeln direkt umsetzen, ist aber numerisch nicht stabil und weicht von in R-Funktionen implementierten Rechnungen ab (Bates, 2004).

```

> Xr1 <- X0                                     # Designmatrix
> Pr1 <- P0                                     # Projektion

# Test Prädiktor 1: umfassenderes Modell
> Xu1 <- X[, c(1, 2)]                           # Designmatrix
> Pu1 <- Xu1 %*% solve(t(Xu1) %*% Xu1) %*% t(Xu1)   # Projektion
> B1  <- t(Y) %*% (Pu1 - Pr1) %*% Y           # Matrix B1

# Test Prädiktor 2: eingeschränktes Modell (= umfass. Modell Präd. 1)
> Xr2 <- Xu1                                     # Designmatrix
> Pr2 <- Pu1                                     # Projektion

# Test Prädiktor 2: umfassenderes Modell
> Xu2 <- X[, c(1, 2, 3)]                         # Designmatrix
> Pu2 <- Xu2 %*% solve(t(Xu2) %*% Xu2) %*% t(Xu2)   # Projektion
> B2  <- t(Y) %*% (Pu2 - Pr2) %*% Y           # Matrix B2

# Test Prädiktor 3: eingeschränktes Modell (= umfass. Modell Präd. 2)
> Xr3 <- Xu2                                     # Designmatrix
> Pr3 <- Pu2                                     # Projektion

# Test Prädiktor 3: umfassenderes Modell (hier = vollst. Modell)
> Xu3 <- X                                       # Designmatrix
> Pu3 <- Pf                                      # Projektion
> B3  <- t(Y) %*% (Pu3 - Pr3) %*% Y           # Matrix B3

```

Mit Hilfe der Matrizen B_j und \mathbf{W} können die Teststatistiken für den Test jedes Prädiktors berechnet werden, was hier nur für den ersten Prädiktor gezeigt werden soll. Die Ergebnisse stimmen mit der Ausgabe von `summary(manova(...))` in Abschn. 12.5 überein.

```

> (WL1 <- det(WW) / det(B1 + WW))                # Wilks' Lambda
[1] 0.1104243

# Roys Maximalwurzel
> (RLRl1 <- max(eigen(solve(WW) %*% B1)$values))    # lambda
[1] 8.055978

> (RLRt1 <- max(eigen(solve(B1 + WW) %*% B1)$values))  # theta
[1] 0.8895757

# Pillai-Bartlett-Spur: hier gleich Roys Maximalwurzel theta
> (PBT1 <- sum(diag(solve(B1 + WW) %*% B1)))
[1] 0.8895757

# Hotelling-Lawley-Spur: hier gleich Roys Maximalwurzel lambda
> (HLT1 <- sum(diag(solve(WW) %*% B1)))
[1] 8.055978

```

12.9.9 Beispiel: Einfaktorielle MANOVA

Das Ergebnis der einfaktoriellen MANOVA in Abschn. 12.7.1 lässt sich nun mit dem in Abschn. 12.9.6 und 12.9.7 dargestellten Vorgehen manuell prüfen. Im Beispiel sollen Daten von zwei AVn (Datenmatrix Y_{m1}) in drei Bedingungen (Faktor IV_{man}) vorliegen. Zunächst sind die Designmatrizen und Projektionen für das eingeschränkte H_0 -Modell sowie für das umfassendere H_1 -Modell zu erstellen.

```
# vollständiges Modell: ursprüngliche Inzidenzmatrix X*p
> XstarP <- cbind(as.numeric(IVman == 1),           # 1. Indikatorvariable
+                   as.numeric(IVman == 2),           # 2. Indikatorvariable
+                   as.numeric(IVman == 3))         # 3. Indikatorvariable

# vollständiges Modell für Treatment-Kontraste
> Ct   <- contr.treatment(ncol(XstarP))          # Kontrastmatrix C
> Xpm1 <- XstarP %*% Ct                          #  $X_{p-1} = X * p * C$ 
> X    <- cbind(1, Xpm1)                         # reduzierte Designmatrix [1|Xp-1]
> XR   <- model.matrix(~ IVman)                  # Designmatrix aus R
> all.equal(X, XR, check.attributes=FALSE)        # Vergleich
[1] TRUE

# orthogonale Projektion auf durch Designmatrix aufgespannten Raum
> Pf <- X %*% solve(t(X) %*% X) %*% t(X)
> Pu <- Pf                                     # hier:  $H_1$ -Modell = vollst. Modell,  $P_u = P_f$ 

# Gesamt-Nullmodell: Designmatrix = 1-Vektor
> X0 <- X[, 1, drop=FALSE]

# orthogonale Projektion auf durch Gesamt-Nullmodell aufgespannten Raum
> P0 <- X0 %*% solve(t(X0) %*% X0) %*% t(X0)
> Pr <- P0                                     # hier:  $H_0$ -Modell = Gesamt-0-Modell,  $P_r = P_0$ 
```

Die Kontrastschätzungen $\hat{B} = \mathbf{X}^+ \mathbf{Y}$ bestehen bei den hier verwendeten Treatment-Kontrasten für jede AV l aus dem Mittelwert der ersten Gruppe ($\hat{\beta}_{0,l} = M_{1,l}$) sowie aus den Abweichungen der verbleibenden Gruppenmittel zu $M_{1,l}$ ($\hat{\beta}_{j,l} = M_{j,l} - M_{1,l}$ mit $j = 2, \dots, p$).

```
# Parameter- / Kontrastschätzungen für Treatment-Kontraste
> (Bt <- coef(lm(Ym1 ~ IVman, contrasts=list(IVman=contr.treatment))))
     [,1]    [,2]
(Intercept) -3.266667  5.60
IVman2       5.626667 -1.56
IVman3       2.916667 -6.00

# Kontrolle: teile Datenmatrix nach Gruppen auf und berechne Zentroide
> splDat <- split(data.frame(Ym1), IVman)
> (Mj      <- t(sapply(splDat, colMeans)))
     X1      X2
1 -3.266667  5.60
```

```

2 2.360000 4.04
3 -0.350000 -0.40

> Mj[2, ] - Mj[1, ]                                # Abweichung M2 - M1 = beta2
   X1          X2
5.626667 -1.560000

> Mj[3, ] - Mj[1, ]                                # Abweichung M3 - M1 = beta3
   X1          X2
2.916667 -6.000000

```

Bei Treatment-Kontrasten stimmen die Schätzungen der ursprünglichen Parameter in $\hat{\boldsymbol{\beta}}_p^* = \mathbf{L}\hat{\boldsymbol{\beta}}_{p-1}$ für jede Zeile $j = 2, \dots, p$ mit jenen in $\hat{\boldsymbol{\beta}}$ überein, während durch die Nebenbedingung $\mathbf{v} = (1, 0, \dots, 0)^t$ für die Parameter in der ersten Zeile $\hat{\beta}_{1,l}^* = 0$ gilt.

```

> (BstarPt <- Ct %*% Bt[-1, ])                  # B*p
   [,1]      [,2]
1 0.000000 0.00
2 5.626667 -1.56
3 2.916667 -6.00

```

Bei der Effektcodierung bestehen die Kontrastschätzungen in $\hat{\boldsymbol{\beta}}$ für jede AV l aus dem ungewichteten Gesamtmittelwert ($\hat{\beta}_{0,l} = M_l = \frac{1}{p} \sum_j M_{j,l}$) sowie aus den Abweichungen der ersten $p-1$ Gruppenmittel zu M_l ($\hat{\beta}_{j,l} = M_{j,l} - M_l$ mit $j = 1, \dots, p-1$).

```

# Parameter- / Kontrastschätzungen für Effektcodierung
> (Be <- coef(lm(Ym1 ~ IVman, contrasts=list(IVman=contr.sum))))
   [,1]      [,2]
(Intercept) -0.4188889 3.08
IVman1       -2.8477778 2.52
IVman2        2.7788889 0.96

> colMeans(Mj)                                     # ungew. Gesamtmittel = beta0
   X1          X2
-0.4188889 3.0800000

> scale(Mj, colMeans(Mj), scale=FALSE)  # Mj.l - M1 = beta1, beta2
   X1          X2
1 -2.8477778 2.52
2  2.7788889 0.96
3  0.06888889 -3.48

```

Die Schätzungen der ursprünglichen Parameter in $\hat{\boldsymbol{\beta}}_p^*$ stimmen bei der Effektcodierung für jede Zeile $j = 1, \dots, p-1$ mit jenen in $\hat{\boldsymbol{\beta}}$ überein. Die Nebenbedingung $\mathbf{v} = \mathbf{1}$ legt für jede AV l die $\hat{\beta}_{p,l}^*$ dadurch fest, dass die Beziehung $\sum_{j=1}^p \hat{\beta}_{j,l}^* = 0$ gelten muss, was zu $\hat{\beta}_{p,l}^* = -\sum_{j=1}^{p-1} \hat{\beta}_{j,l}^*$ führt.

```
> Ce <- contr.sum(ncol(XstarP))           # Kontrastmatrix Effektcodierung
> (BstarPe <- Ce %*% betaE[-1 , ])      # B*p
[1] [,1] [,2]
1 -2.84777778  2.52
2  2.77888889  0.96
3  0.06888889 -3.48

> colSums(BstarPe)                      # Summe über beta*p = 0
[1] 0.000000e+00 1.110223e-16
```

Bei der Parametrisierung ohne β_0 in der Rolle des theoretischen Zentroids μ (cell means Modell, vgl. Abschn. 12.9.2, Fußnote 48) erhalten die ursprünglichen Parameter in B_p^* die Bedeutung der Gruppenzentroide μ_j und werden entsprechend über die Gruppenmittelwerte geschätzt. Für jede AV l gilt also $\hat{\beta}_{j,l}^* = M_{j,l}$, zudem stimmt B_p^* mit B_p überein.

```
# Parameterschätzungen B = B* für cell means Modell
> (Bcm <- coef(lm(Ym1 ~ IVman - 1)))
[1] [,1] [,2]
IVman1 -3.266667  5.60
IVman2  2.360000  4.04
IVman3 -0.350000 -0.40
```

Die Vorhersage $\hat{Y}_f = P_f Y$ des vollständigen Modells liefert für eine Person in der Gruppe j für jede AV l den zugehörigen Gruppenmittelwert $M_{j,l}$.

```
> Yhat <- Pf %*% Ym1                  # Vorhersage vollst. Modell
> unique(Yhat)                         # vorkommende Werte
[1,] [,1] [,2]
[1,] -3.266667  5.60
[2,]  2.360000  4.04
[3,] -0.350000 -0.40
```

Die für den inferenzstatistischen Test notwendigen Matrizen T , B und W ergeben sich aus den ermittelten Projektionen $P_r = P_0$ und $P_u = P_f$ jeweils auf den Unterraum, der durch die Designmatrix des eingeschränkten und des umfassenderen Modells aufgespannt wird.

```
> Id <- diag(sum(Nj))                 # n x n Einheitsmatrix I
> BB <- t(Ym1) %*% (Pu-Pr) %*% Ym1   # B
> WW <- t(Ym1) %*% (Id-Pf) %*% Ym1    # W
> TT <- t(Ym1) %*% (Id-PO) %*% Ym1    # T

# B: SSP-Matrix der durch Vorhersagedifferenzen ersetzen Daten
> all.equal(BB, (sum(Nj)-1) * cov((Pu-Pr) %*% Ym1))
[1] TRUE

# W: SSP-Matrix der durch Differenz zum jeweiligen
# Gruppenzentroid ersetzen Daten
> all.equal(WW, (sum(Nj)-1) * cov((Id-Pf) %*% Ym1))
```

```
[1] TRUE

# T: SSP-Matrix der Residuen des Nullmodells sowie der Daten
> all.equal(TT, (sum(Nj)-1) * cov((Id-P0) %*% Ym1))
[1] TRUE

> all.equal(TT, (sum(Nj)-1) * cov(Ym1))           # SSP-Matrix Daten
[1] TRUE

> all.equal(TT, BB + WW)                          # Kontrolle: T = B + W
[1] TRUE
```

Mit Hilfe der Matrizen B und W können die Teststatistiken berechnet werden, die mit der Ausgabe von `summary(manova(...))` in Abschn. 12.7.1 übereinstimmen.

```
> (WL <- det(WW) / det(BB + WW))          # Wilks' Lambda
[1] 0.4201068

# Roys Maximalwurzel
> (RLR1 <- max(eigen(solve(WW) %*% BB)$values))      # lambda
[1] 0.6956387

> (RLRt <- max(eigen(solve(BB + WW) %*% BB)$values))    # theta
[1] 0.4102517

> (PBT <- sum(diag(solve(BB + WW) %*% BB)))    # Pillai-Bartlett-Spur
[1] 0.6979024

> (HLT <- sum(diag(solve(WW) %*% BB)))        # Hotelling-Lawley-Spur
[1] 1.099444
```

12.9.10 Beispiel: Zweifaktorielle MANOVA

Das Ergebnis der zweifaktoriellen MANOVA in Abschn. 12.7.2 lässt sich nun ebenfalls mit dem in Abschn. 12.9.6 und 12.9.7 dargestellten Vorgehen manuell prüfen. Es sollen Daten auf zwei AVn (Datenmatrix Y_{m2}) in 3×2 Bedingungen (Faktoren IV_1 und IV_2) vorliegen. Zunächst sind die ursprünglichen Inzidenzmatrizen $X_1^*, X_2^*, X_{1 \times 2}^*$ zu erstellen. Ihr jeweiliges Produkt mit der zugehörigen Kontrastmatrix L geht in die Designmatrizen der eingeschränkten und umfassenderen Modelle ein, die den Tests der drei Hypothesen (zwei Haupteffekte und Interaktionseffekt) zugrunde liegen.

```
# ursprüngliche Inzidenzmatrizen
> Xstar1 <- cbind(as.numeric(IV1 == 1),           # UV 1: X*1
+                   as.numeric(IV1 == 2),
+                   as.numeric(IV1 == 3))

> Xstar2 <- cbind(as.numeric(IV2 == 1),           # UV 2: X*2
```

Kapitel 12 Multivariate Verfahren

```

+           as.numeric(IV2 == 2))

# Interaktion: alle paarweisen Produkte der Spalten von X*1 und X*2: X*1x2
> Xstar12 <- cbind(as.numeric(IV1 == 1) * as.numeric(IV2 == 1),
+                   as.numeric(IV1 == 2) * as.numeric(IV2 == 1),
+                   as.numeric(IV1 == 3) * as.numeric(IV2 == 1),
+                   as.numeric(IV1 == 1) * as.numeric(IV2 == 2),
+                   as.numeric(IV1 == 2) * as.numeric(IV2 == 2),
+                   as.numeric(IV1 == 3) * as.numeric(IV2 == 2))

# Kontrastmatrizen: Treatment-Kontraste
> C1 <- contr.treatment(ncol(Xstar1))          # UV 1
> C2 <- contr.treatment(ncol(Xstar2))          # UV 2
> C12 <- kronecker(C2, C1)                      # Interaktion

# reduzierte Inzidenzmatrizen
> X1 <- Xstar1  %*% C1                         # UV 1: X1 = X*1 * C1
> X2 <- Xstar2  %*% C2                         # UV 2: X2 = X*2 * C2
> X12 <- Xstar12 %*% C12                        # Interaktion: X*1x2 * C1x2

# vollständiges Modell: Designmatrix für identifizierbare Parameter
> X <- cbind(1, X1, X2, X12)                    # X = [1|X1|X2|X1x2]
> XR <- model.matrix(~ IV1*IV2)                 # Designmatrix aus R
> all.equal(X, XR, check.attributes=FALSE)       # Vergleich
[1] TRUE

```

Im Gesamt-Nullmodell der zweifaktoriellen Varianzanalyse sind alle Parameter bis auf β_0 gleich 0, als Designmatrix \mathbf{X}_0 bleibt also der Vektor 1. Es folgt die Berechnung der zugehörigen Projektion \mathbf{P}_0 sowie der zum vollständigen Modell mit Designmatrix \mathbf{X}_f gehörenden Projektion \mathbf{P}_f . Aus der Projektion auf das orthogonale Komplement des jeweils von den Spalten von \mathbf{X}_0 und \mathbf{X}_f aufgespannten Unterraumes ergeben sich die Matrizen \mathbf{T} und \mathbf{W} .

```

# Gesamt-Nullmodell
> X0 <- X[, 1, drop=FALSE]                     # Designmatrix X0 = 1-Vektor

# Nullmodell: orthogonale Projektion auf durch X0 aufgespannten Raum
> P0 <- X0 %*% solve(t(X0) %*% X0) %*% t(X0)

# vollst. Modell: orth. Proj. auf durch Designmatrix aufgespannten Raum
> Pf <- X %*% solve(t(X) %*% X) %*% t(X)

# Matrizen T und W für Teststatistiken
> Id <- diag(nrow(Ym2))                         # n x n Einheitsmatrix I
> TT <- t(Ym2) %*% (Id - P0) %*% Ym2          # T
> WW <- t(Ym2) %*% (Id - Pf) %*% Ym2          # W

```

Die Vorhersage $\hat{\mathbf{Y}}_f = \mathbf{P}_f \mathbf{Y}$ des vollständigen Modells liefert für eine Person in der Bedingungskombination jk beider UVn für jede AV l den Gruppenmittelwert $M_{jk.l}$.

```
> Yhat <- Pf %*% Ym2                                # Vorhersage vollständiges Modell
> unique(Yhat)                                     # vorkommende Werte
[1]      [,1]      [,2]
[1,] -3.26666667  5.600000
[2,]  2.36000000  4.040000
[3,] -0.35000000 -0.400000
[4,]  0.06666667  4.666667
[5,]  3.68000000  7.960000
[6,]  4.05000000 -0.400000

# Kontrolle: teile Datenmatrix nach Gruppen auf und berechne Zentroide
> splDat <- split(data.frame(Ym2), list(IV1, IV2))
> t(sapply(splDat, colMeans))
      X1          X2
1.1 -3.26666667  5.600000
2.1  2.36000000  4.040000
3.1 -0.35000000 -0.400000
1.2  0.06666667  4.666667
2.2  3.68000000  7.960000
3.2  4.05000000 -0.400000
```

Schließlich muss für jede der drei Hypothesen das zugehörige Paar aus eingeschränktem H_0 -Modell und umfassenderem H_1 -Modell mit zugehörigen Designmatrizen \mathbf{X}_r und \mathbf{X}_u sowie ihren orthogonalen Projektionen \mathbf{P}_r und \mathbf{P}_u berechnet werden. Dies soll hier für Quadratsummen vom Typ I geschehen. Aus der Differenz beider Projektionen ergibt sich für jeden Effekt die Matrix \mathbf{B}_j , die als Verallgemeinerung der univariaten Effekt-Quadratsumme in die zugehörigen Teststatistiken eingeht.

```
# Test UV 1: eingeschränktes Modell (= Gesamt-Nullmodell)
> Xr1 <- X0                                         # Designmatrix
> Pr1 <- P0                                         # Projektion

# Test UV 1: umfassenderes Modell
> Xu1 <- X[, c(1, 2, 3)]                           # Designmatrix
> Pu1 <- Xu1 %*% solve(t(Xu1) %*% Xu1) %*% t(Xu1) # Projektion
> B1 <- t(Ym2) %*% (Pu1-Pr1) %*% Ym2              # Matrix B1

# Test UV 2: eingeschränktes Modell (= umfassenderes Modell UV 1)
> Xr2 <- Xu1                                         # Designmatrix
> Pr2 <- Pu1                                         # Projektion

# Test UV 2: umfassenderes Modell
> Xu2 <- X[, c(1, 2, 3, 4)]                         # Designmatrix
> Pu2 <- Xu2 %*% solve(t(Xu2) %*% Xu2) %*% t(Xu2) # Projektion
> B2 <- t(Ym2) %*% (Pu2-Pr2) %*% Ym2              # Matrix B2
```

```
# Test Interaktion: eingeschränktes Modell (= umfass. Modell UV 2)
> Xr12 <- Xu2                                     # Designmatrix
> Pr12 <- Pu2                                     # Projektion

# Test Interaktion: umfassenderes Modell (hier = vollst. Modell)
> Xu12 <- X                                       # Designmatrix
> Pu12 <- Pf                                      # Projektion
> B12   <- t(Ym2) %*% (Pu12 - Pr12) %*% Ym2      # Matrix B12

# Kontrolle: Verallgemeinerung der Quadratsummenzerlegung
# gilt allgemein nur für QS vom Typ I, sonst nur bei gleichen Njk
> all.equal(TT, B1 + B2 + B12 + WW)
[1] TRUE
```

Mit Hilfe der Matrizen B_1 , B_2 , $B_{1 \times 2}$ und W können die Teststatistiken für den Test jedes Effekts berechnet werden, was hier nur für den ersten Haupteffekt gezeigt werden soll. Die Ausgabe stimmt mit jener von `summary(manova(...))` in Abschn. 12.7.2 überein.

```
> (WL1 <- det(WW) / det(B1 + WW))                  # Wilks' Lambda
[1] 0.3846785

# Roys Maximalwurzel
> (RLR11 <- max(eigen(solve(WW) %*% B1)$values))    # lambda
[1] 1.042646

> (RLRt1 <- max(eigen(solve(B1 + WW) %*% B1)$values))  # theta
[1] 0.5104389

> (PBT1 <- sum(diag(solve(B1 + WW) %*% B1)))       # Pillai-Bartlett-Spur
[1] 0.7246769

> (HLT1 <- sum(diag(solve(WW) %*% B1)))            # Hotelling-Lawley-Spur
[1] 1.315296
```

Kapitel 13

Vorhersagegüte prädiktiver Modelle

Da empirische Daten fehlerbehaftet sind, bezieht die Anpassung eines statistischen Modells immer auch die Messfehler mit ein, die Parameterschätzungen orientieren sich daher zu stark an den zufälligen Besonderheiten der konkreten Stichprobe (*overfitting*). Die Güte der Passung des Modells lässt sich als Funktion $f(\cdot)$ der Abweichungen $E = Y - \hat{Y}$ der Modellvorhersage \hat{Y} zu den tatsächlichen Werten der vorhergesagten Variable Y quantifizieren. Genauer soll $\hat{Y}_{X,Y}(X')$ die folgende Vorhersage bezeichnen: Zunächst wird ein Modell an einer Stichprobe mit Werten für Prädiktoren X und Zielvariable Y (Kriterium) angepasst. In die Vorhersagegleichung mit den Parameterschätzungen dieses Modells werden dann (potentiell andere) Prädiktorwerte X' eingesetzt, um die Vorhersage \hat{Y} zu berechnen, die mit den tatsächlichen Beobachtungen Y' zu vergleichen sind. $f(E)$ ist die *Verlustfunktion*, die alle individuellen absoluten Abweichungen e_i auf einen Gesamtwert für die Vorhersagegenauigkeit abbildet.

Angewendet auf die zur Modellanpassung verwendete *Trainingsstichprobe* selbst ($X' = X$) soll $f(E)$ hier als *Trainingsfehler* (auch Resubstitutionsfehler) bezeichnet werden, angewendet auf andere *Teststichproben* aus derselben Grundgesamtheit ($X' \neq X$) als *Vorhersagefehler*. Die Passung des Modells für die Trainingsstichprobe ist dabei im Mittel besser als für andere Teststichproben aus derselben Grundgesamtheit. Der Trainingsfehler ist dahingehend verzerrt, dass er den Vorhersagefehler systematisch unterschätzt, also zu optimistisch bzgl. der Generalisierbarkeit des angepassten Modells ist. Die Größe des Vorhersagefehlers liefert auch einen Anhaltspunkt für die Auswahl eines bestimmten Modells aus mehreren möglichen (vgl. Abschn. 6.3.3).

Die folgenden Abschnitte stellen Kreuzvalidierung und Bootstrapping für eine möglichst unverzerrte Schätzung des Vorhersagefehlers vor (Hastie, Tibshirani & Friedman, 2009; James, Witten, Hastie & Tibshirani, 2013; Kuhn & Johnson, 2013). Beide Methoden sind insofern Resampling-Verfahren (vgl. Kap. 11), als sie aus Daten einer gegebenen Basisstichprobe mehrfach neue Stichproben erstellen und mit ihnen den Vorhersagefehler schätzen.

13.1 Kreuzvalidierung linearer Regressionsmodelle

Für die Kreuzvalidierung ist eine vorliegende Gesamtstichprobe in zwei komplementäre Teilmengen zu partitionieren: Die Trainingsstichprobe liefert die Datenbasis für die Parameterschätzung. Diese Modellanpassung wird dann auf die verbleibende Teststichprobe angewendet, indem die Werte der Kovariaten X' aus der Teststichprobe in die ermittelte Vorhersagegleichung eingesetzt werden. Als Verlustfunktion $f(E)$ dient etwa die mittlere Fehlerquadratsumme, im Falle einer linearen Regression der Standardschätzfehler.

13.1.1 *k*-fache Kreuzvalidierung

Für eine bessere Beurteilung der Vorhersagegenauigkeit sollte die dargestellte Prüfung mehrfach mit wechselnden Trainings- und Teststichproben durchgeführt werden. Häufig wird hierzu die Ausgangsstichprobe in k disjunkte Teilmengen partitioniert. Jede von ihnen dient daraufhin reihum als Teststichprobe, während die jeweils verbleibenden $k - 1$ Stichproben gemeinsam die Trainingsstichprobe bilden. Der Mittelwert der Verlustfunktion über die k Wiederholungen dient dazu, die Generalisierbarkeit der Parameterschätzung insgesamt zu beurteilen.¹ Häufig empfohlene Werte für k sind 5 oder 10, für höhere Werte ist der steigende numerische Aufwand zur Kreuzvalidierung sehr umfassender Modelle zu berücksichtigen. Bei $k = 2$ mit zwei gleich großen Teilstichproben handelt es sich um die doppelte Kreuzvalidierung.²

Das Paket **boot** (vgl. Abschn. 11.1) stellt mit `cv.glm()` eine Funktion für die k -fache Kreuzvalidierung verallgemeinerter linearer Modelle (vgl. Kap. 8) bereit.

```
> cv.glm(data=<Datensatz>, glmfit=<glm-Modell>, K=<Anzahl>,
+         cost=<Verlustfunktion>)
```

Für `data` ist der Datensatz zu übergeben, aus dem die Variablen eines verallgemeinerten linearen Modells stammen, das unter `glmfit` zu nennen ist. Da die lineare Regression ein Spezialfall eines solchen Modells ist, kann `cv.glm()` auch für sie zum Einsatz kommen. Die Regression ist hierfür lediglich mit `glm()` zu formulieren (vgl. Abschn. 8.1.6). Mit `K` lässt sich die gewünschte Anzahl zufälliger Partitionen für die Aufteilung in Trainings- und Teststichproben wählen. In der Voreinstellung ist die Verlustfunktion die mittlere Fehlerquadratsumme, kann aber über `cost` auf eine selbst definierte Funktion geändert werden (vgl. Abschn. 13.2).

Das Ergebnis von `cv.glm()` ist eine Liste, die in der Komponente `delta` den Kreuzvalidierungsfehler (CVE, cross validation error) beinhaltet, in der Voreinstellung also den Mittelwert der k mittleren Fehlerquadratsummen in der Teststichprobe. Das zweite Element von `delta` berücksichtigt dabei die Korrektur einer aus der Wahl von k herrührenden Verzerrung.

```
# passe lineare Regression mit glm() an
> regrDf <- data.frame(weight, height, age, sport)      # Datensatz
> glmFit <- glm(weight ~ height + age + sport, data=regrDf,
+                  family=gaussian(link="identity"))

> library(boot)                                         # für cv.glm()
> k     <- 3                                           # Anzahl Partitionen
> kfCV <- cv.glm(regrDf, glmfit=glmFit, K=k)       # k-fache Kreuzvalid.
> kfCV$delta                                         # Kreuzvalidierungsfehler und Korrektur
10.52608 10.38042
```

¹Die k -fache Kreuzvalidierung eines linearen Regressionsmodells ist für $k = n \left(1 - \frac{1}{\ln(n)-1}\right)$ asymptotisch äquivalent zum Informationskriterium BIC (vgl. Abschn. 6.3.3).

²Die *stratifizierte* k -fache Kreuzvalidierung sorgt bei der Einteilung in Teilmengen dafür, dass die Verteilung von Y in den Partitionen ähnlich ist. Für kontinuierliche Y führt dies zu einem annähernd konstanten Mittelwert von Y in den Partitionen. Für kategoriale Y bleibt so der Anteil der Kategorien weitgehend gleich. Für eine Umsetzung vgl. die Pakete **caret** (Kuhn, 2014) oder **ipred** (Peters & Hothorn, 2013).

Für die manuelle Berechnung ist zunächst eine eigene Funktion zu erstellen, die auf Basis des logischen Indexvektors einer Teststichprobe eine einzelne Kreuzvalidierung des gegebenen Regressionsmodells durchführt (vgl. Abschn. 15.2).

```
# einzelne Kreuzvalidierung gegeben logische Indizes der Teststichprobe
> doCV <- function(idxTst) {
+   # passe Regression in Trainingsstichprobe an
+   fitTrn <- lm(weight ~ height + age + sport, regrDf, subset=!idxTst)
+
+   # berechne für Teststichprobe Vorhersage aus angepasstem Modell
+   predTst <- predict(fitTrn, regrDf[idxTst, ])
+
+   # mittlere Fehlerquadratsumme für Teststichprobe
+   mean((predTst - weight[idxTst])^2)
+ }
```

Daraufhin muss die gesamte Stichprobe vom Umfang n in k zufällige Gruppen partitioniert werden, die dann der Reihe nach als Teststichprobe dienen.

```
> tstGrp <- (sample(1:N, N, replace=FALSE) %% k) + 1      # Gruppen
> (tst1 <- doCV(tstGrp == 1))    # Kreuzvalidierung 1. Teststichprobe
[1] 11.96789

> (tst2 <- doCV(tstGrp == 2))    # Kreuzvalidierung 2. Teststichprobe
[1] 10.45271

> (tst3 <- doCV(tstGrp == 3))    # Kreuzvalidierung 3. Teststichprobe
[1] 11.85106

> mean(c(tst1, tst2, tst3))      # Kreuzvalidierungsfehler
[1] 11.42389
```

13.1.2 Leave-One-Out Kreuzvalidierung

Ein Spezialfall der k -fachen Kreuzvalidierung ist die *Leave-One-Out*-Kreuzvalidierung (LOOCV) für $k = n$.³ Hier dient nacheinander jede Einzelbeobachtung separat als Teststichprobe für ein Modell, das an der Trainingsstichprobe aus jeweils allen übrigen Beobachtungen angepasst wurde. Im Fall der linearen Regression lässt sich der Kreuzvalidierungsfehler hier numerisch effizient als Mittelwert der *PRESS*-Residuen $\frac{1}{n} \sum_i \left(\frac{e_i}{1-h_i} \right)^2$ (*predicted residual error sum of squares*) direkt berechnen. Dabei ist e_i das i -te Residuum $y_i - \hat{y}_i$ und h_i der Hebelwert der Beobachtung i (vgl. Abschn. 6.5.1).

```
# LOOCV-Kreuzvalidierungsfehler inkl. korrigierter Variante
> LOOCV <- cv.glm(data=regrDf, glmfit=glmFit, K=N)
> LOOCV$delta
```

³Sie ist asymptotisch äquivalent zum Informationskriterium AIC des Regressionsmodells (vgl. Abschn. 6.3.3).

1	1
10.59404	10.58995

```
# Kreuzvalidierungsfehler hier = Mittelwert PRESS-Residuen
> lmFit <- lm(weight ~ height + age + sport)
> PRESS <- (residuals(lmFit) / (1 - hatvalues(lmFit)))^2
> mean(PRESS)
[1] 10.59404
```

Bei der Kontrolle wird die zuvor selbst erstellte Funktion `doCV()` mittels `sapply()` für jeden Index der Stichprobe aufgerufen.

```
> idx <- seq(length.out=N)           # Indizes aller Beobachtungen
> res <- sapply(idx, function(x) { doCV(idx == x) } )      # LOOCV
> mean(res)                      # Kreuzvalidierungsfehler
[1] 10.59404
```

Für penalisierte Regressionsmodelle und verallgemeinerte additive Modelle (vgl. Abschn. 6.6) ist die Berechnung der Hat-Matrix H – und damit der Diagonaleinträge h_i – aufwendig. In diesen Fällen lässt sich der Kreuzvalidierungsfehler über die verallgemeinerte Kreuzvalidierung (GCV) mit $\frac{1}{n} \sum_i \left(\frac{e_i}{1 - \text{Spur}(H)/n} \right)^2$ approximieren. Hier ist $\text{Spur}(H)$ die Anzahl zu schätzender Parameter.

13.2 Kreuzvalidierung verallgemeinerter linearer Modelle

Bei der Kreuzvalidierung verallgemeinerter linearer Modelle (vgl. Kap. 8) ist zu beachten, dass die Wahl der Verlustfunktion als Gütemaß der Vorhersagegenauigkeit für diskrete Variablen Besonderheiten mit sich bringt. Im Fall der logistischen Regression (vgl. Abschn. 8.1) wäre es etwa naheliegend, die Vorhersagegüte als Anteil der korrekt vorhergesagten Treffer umzusetzen. Tatsächlich ist dieses Maß weniger gut geeignet, da mit ihm nicht das wahre statistische Modell den kleinsten Vorhersagefehler besitzt. Verlustfunktionen, die durch das wahre Modell minimiert werden, heißen *proper score*.

Das Argument `cost` von `cv.glm()` für selbst definierte Verlustfunktionen erwartet eine Funktion, die auf Basis je eines Vektors der beobachteten Werte Y und der vorhergesagten Werte \hat{Y} ein Abweichungsmaß zurückgibt. `cv.glm()` verwendet beim Aufruf von `cost()` für Y den Vektor `<glm-Modell>$y` und für \hat{Y} `predict(<glm-Modell>, type="response")`.

Für die logistische Regression ist der Brier-Score B eine geeignete Verlustfunktion. Für jede Beobachtung i berücksichtigt b_i die vorhergesagte Wahrscheinlichkeit $\hat{p}_{ij} = \hat{p}(y_i = j)$ jeder Kategorie j beim Vergleich mit der tatsächlich vorliegenden Kategorie y_i . Hier soll \hat{p}_i kurz für die vorhergesagte Trefferwahrscheinlichkeit $p(y_i = 1)$ stehen, und es gilt $1 - p_i = p(y_i = 0)$.

$$b_i = \begin{cases} (1 - \hat{p}_i)^2 + (0 - (1 - \hat{p}_i))^2 & \text{falls } y_i = 1 \\ (1 - (1 - \hat{p}_i))^2 + (0 - \hat{p}_i)^2 & \text{falls } y_i = 0 \end{cases}$$

Der Brier-Score für eine Stichprobe vom Umfang n ist dann $B = \frac{1}{n} \sum_i b_i$. Mit $k = 2$ Kategorien und dem Kronecker δ als Indikatorfunktion lässt sich b_i auch verkürzt schreiben, wobei $\delta_{ij} = 1$ ist, wenn $y_i = j$ gilt und $\delta_{ij} = 0$ sonst.

$$b_i = \sum_{j=1}^k (\delta_{ij} - \hat{p}_{ij})^2$$

Als Beispiel seien die Daten aus Abschnitt 8.1 zur logistischen Regression betrachtet.

```
> glmLR <- glm(postFac ~ DVpre, family=binomial(link="logit"),
+                 data=dfAncova)

# Verlustfunktion für Brier-Score
> brierA <- function(y, pHat) {
+   mean(((y == 1) * pHat)^2 + ((y == 0) * (1-pHat))^2)
+ }

> library(boot)                                     # für cv.glm()
> B1 <- cv.glm(data=regDf, glmfit=glmFit, cost=brierA, K=10)
> B1$delta
[1] 0.5528585 0.5523104
```

Die genannte Definition des Brier-Score lässt sich unmittelbar auf Situationen mit $k > 2$ Kategorien verallgemeinern, etwa auf die multinomiale Regression oder die Diskriminanzanalyse, wo er ebenfalls ein proper score ist. Speziell für die logistische Regression führt auch die folgende vereinfachte Variante zu einem proper score, sie berücksichtigt nur die tatsächlich beobachtete Kategorie. Diese Verlustfunktion $b_i = (y_i - \hat{p}_i)^2$ entspricht der Voreinstellung von `cost` in `cv.glm()`.

```
# Verlustfunktion für Brier-Score - vereinfachte Variante
> brierB <- function(y, pHat) {
+   mean((y-pHat)^2)
+ }

> B2 <- cv.glm(data=regDf, glmfit=glmFit, cost=brierB, K=10)
> B2$delta
[1] 0.1787110 0.1773381
```

Eine Alternative zum Brier-Score basiert auf der logarithmierten geschätzten Wahrscheinlichkeit für die tatsächlich beobachtete Kategorie j von Y . Die Verlustfunktion $-2 \cdot \sum_i \ln \hat{p}_{ij}$ ist äquivalent zu Devianz-Residuen und führt ebenfalls zu einem proper score, der sich auch auf andere verallgemeinerte lineare Modelle anwenden lässt.

13.3 Bootstrap-Vorhersagefehler

Bootstrapping (vgl. Abschn. 11.1) liefert weitere Möglichkeiten zur unverzerrten Schätzung des Vorhersagefehlers prädiktiver Modelle, von denen hier die einfache Optimismus-Korrektur

vorgestellt wird. Für weitere Ansätze wie den .632 und .632+ bootstrap vgl. Hastie et al. (2009) und für eine Implementierung die Pakete `caret` oder `ipred`.

Für die Optimismus-Berechnung wird die (negative) systematische Verzerrung des Trainingsfehlers über viele Replikationen der Basisstichprobe hinweg geschätzt und danach als Bias-Korrektur vom ursprünglichen Trainingsfehler subtrahiert. Zur Erläuterung der notwendigen Schritte sei folgende Notation vereinbart:

- $\mathbb{E}[Y - \hat{Y}_{X_0, Y_0}(X)]$ ist der wahre Vorhersagefehler eines an den Beobachtungen X_0, Y_0 der Basisstichprobe angepassten Modells für eine neue Stichprobe mit Beobachtungen X, Y von Personen aus derselben Grundgesamtheit.
- $\bar{E}[Y_0 - \hat{Y}_{X_0, Y_0}(X_0)]$ ist der mittlere Trainingsfehler eines an den Beobachtungen X_0, Y_0 der Basisstichprobe angepassten Modells. $\bar{E}[Y_0 - \hat{Y}_{X_0, Y_0}(X_0)]$ unterschätzt $\mathbb{E}[Y - \hat{Y}_{X_0, Y_0}(X)]$ systematisch, die Differenz wird als *Optimismus* des Trainingsfehlers bezeichnet.
- $\bar{E}[Y_0 - \hat{Y}_{X^*, Y^*}(X_0)]$ ist der mittlere Vorhersagefehler der Bootstrap-Replikationen, deren jeweilige Vorhersagen für die Basisstichprobe berechnet werden. $\bar{E}[Y_0 - \hat{Y}_{X^*, Y^*}(X_0)]$ ist plug-in-Schätzer für $\mathbb{E}[Y - \hat{Y}_{X_0, Y_0}(X)]$.
- $\bar{E}[Y^* - \hat{Y}_{X^*, Y^*}(X^*)]$ ist der mittlere Trainingsfehler der Bootstrap-Replikationen und plug-in-Schätzer für $\bar{E}[Y_0 - \hat{Y}_{X_0, Y_0}(X_0)]$.

Da bootstrapping die Beziehung zwischen Basisstichprobe und Population auf die Beziehung zwischen Replikation und Basisstichprobe überträgt, erfolgt die Bootstrap-Schätzung des wahren Trainingsfehler-Optimismus durch die Differenz der jeweiligen plug-in Schätzer. Die bias-korrigierte Schätzung des Vorhersagefehlers $\hat{\mathbb{E}}$ ist dann die Differenz vom Trainingsfehler der Basisstichprobe und dem (negativen) geschätzten Optimismus.

$$\hat{\mathbb{E}}[Y - \hat{Y}_{X_0, Y_0}(X)] = \bar{E}[Y_0 - \hat{Y}_{X_0, Y_0}(X_0)] - (\bar{E}[Y^* - \hat{Y}_{X^*, Y^*}(X^*)] - \bar{E}[Y_0 - \hat{Y}_{X^*, Y^*}(X_0)])$$

Der Stichprobenumfang im hier verwendeten Beispiel einer logistischen Regression (vgl. Abschn. 8.1) ist gering. In den Replikationen können deshalb auch solche Zusammensetzungen der Beobachtungen auftreten, die zu einer vollständigen Separierbarkeit führen und ungültige Parameterschätzungen liefern (vgl. Abschn. 8.1.7). Mit den in Abschn. 15.2.3 und 15.3.2 vorgestellten Methoden kann diese Situation innerhalb der von `boot()` für jede Replikation aufgerufenen Funktion identifiziert werden, um dann einen fehlenden Wert anstatt der ungültigen Optimismus-Schätzung zurückzugeben. Als Verlustfunktion soll der vereinfachte Brier-Score für die logistische Regression (s. o.) dienen.

```
# Funktion, die Optimismus in einer Replikation schätzt
> getBSB <- function(dat, idx) {
+   op <- options(warn=2) # mache Warnungen zu Fehlern
+   on.exit(par(op))      # bei Verlassen: Option zurücksetzen
+
+   # logistische Regression, bei der Fehler abgefangen werden
+   bsFit <- try(glm(postFac ~ DVpre, family=binomial(link="logit"),
+                   subset=idx, data=dat))
+
+   fail <- inherits(bsFit, "try-error") # Fehler in glm()?
```

```

+ if(fail || !bsFit$converged) { # Fehler | keine Konvergenz
+   return(NA)
+ } else {                                # alles ok
+   # Trainingsfehler Replikation
+   BbsTrn <- brierB(bsFit$y, predict(bsFit, type="response"))
+
+   # Vorhersagefehler Replikation bzgl. Basisstichprobe
+   BbsTst <- brierB(as.numeric(dat$postFac)-1,
+                      predict(bsFit, newdata=dat, type="response"))
+
+   return(BbsTrn - BbsTst)                 # Optimismus Replikation
+ }
+ }
```

Im Anschluss an das bootstrapping sollte geprüft werden, wie oft Konvergenz- oder Separierbarkeitsprobleme aufgetreten sind.

```

> library(boot)                         # für boot()
> nR      <- 999                      # Anzahl Replikationen
> bsRes <- boot(dfAncova, statistic=getBSB, R=nR)
> sum(is.na(bsRes$t))                  # Anzahl problematischer Anpassungen
[1] 5

# Traingsfehler Basisstichprobe
> (Btrain <- brierB(glmLR$y, predict(glmLR, type="response")))
[1] 0.1494605

> (optimism <- mean(bsRes$t, na.rm=TRUE))    # mittlerer Optimismus
[1] -0.02267715

# Bootstrap-Schätzung Optimismus-korrigierter Vorhersagefehler
> (predErr <- Btrain - optimism)
[1] 0.1721376
```

Kapitel 14

Diagramme erstellen

Daten lassen sich in R mit Hilfe einer Vielzahl von Diagrammtypen grafisch darstellen, wobei hier nur auf eine Auswahl der verfügbaren Typen eingegangen werden kann.¹ In R werden zwei Arten von Grafikfunktionen unterschieden: *High-Level*-Funktionen erstellen eigenständig ein komplettes Diagramm inkl. Achsen, während *Low-Level*-Funktionen lediglich ein bestimmtes Element einem bestehenden Diagramm hinzufügen. Einen kurzen Überblick über die Gestaltungsmöglichkeiten vermittelt `demo(graphics)`.

14.1 Grafik-Devices

14.1.1 Aufbau und Verwaltung von Grafik-Devices

Die Ausgabe von Befehlen zum Erstellen einer Grafik kann in verschiedenen Ausgabekanälen, den *devices* erfolgen. Ein device lässt sich wie eine leere Leinwand vorstellen, auf der mit Grafikfunktionen einzelne Inhalte wie mit einem Pinsel eingezeichnet werden. Die Fläche des device ist dabei in drei ineinander verschachtelte Regionen eingeteilt: die gesamte Device-Region mit den äußereren Rändern, die innerhalb dieser Ränder liegende *Figure*-Region und die in diese eingebettete *Plot*-Region, in die die Datenpunkte und andere Grafikelemente eingezeichnet werden (Abb. 14.1). In der Voreinstellung ist ein device ein separates Grafikfenster, es können aber etwa auch Dateien in verschiedenen Grafikformaten als device dienen.

Sofern noch kein Grafikfenster existiert, öffnet es sich mit Eingabe des ersten High-Level-Grafikbefehls automatisch – in RStudio im Plots Tab. In dieses Fenster werden dann alle weiteren Ausgaben grafischer Funktionen hinein gezeichnet, wobei im Fall von High-Level-Funktionen ein ggf. bereits vorhandener Inhalt gelöscht wird. Soll für die Ausgabe einer Grafikfunktion zusätzlich zu bereits bestehenden ein neues, zunächst leeres Fenster geöffnet werden, geschieht dies unter Windows mit

```
> windows(width=<Breite>, height=<Höhe>)
```

Unter MacOS ist `quartz()` und unter Unix/Linux `x11()` der äquivalente Befehl. Unabhängig vom Betriebssystem erzielt `dev.new()` in der Voreinstellung denselben Effekt. Breite und Höhe des Fensters können über die Argumente `width` und `height` in der Einheit inch bestimmt

¹Zudem soll fast ausschließlich das grafische Basissystem, nicht aber fortgeschrittene Alternativen behandelt werden, wie sie etwa durch die Pakete `lattice` (Sarkar, 2008) oder `ggplot2` (Wickham, 2009) realisiert werden (vgl. Abschn. 14.8.3). Ebenso bleiben Funktionen weitgehend ausgespart, die einen interaktiven Umgang mit Grafiken erlauben (vgl. Abschn. 14.8.2). Für eine umfassende Dokumentation vgl. Murrell (2011) und für Online-Beispiele Chang (2013).

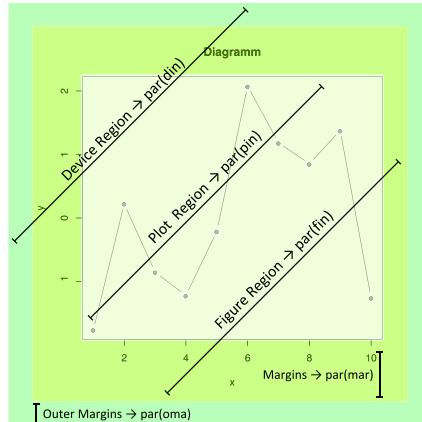


Abbildung 14.1: Regionen und Ränder eines device samt Möglichkeiten, ihre Größe mit `par()` ausgeben und ggf. verändern zu können (vgl. Abschn. 14.3.1)

werden. Bei mehreren geöffneten devices sind alle bis auf eines inaktiv, das im Fenstertitel die Bezeichnung (**ACTIVE**) trägt. Die Bezeichnung bedeutet, dass in dieses device die Ausgabe der folgenden Grafikfunktion gezeichnet wird, während die Inhalte der anderen device unverändert bleiben. Um sich einen Überblick über alle aktuell geöffneten devices zu verschaffen, dient der Befehl `dev.list()`.

```
> dev.new(); dev.new(); dev.new()
> dev.list()
windows windows windows
      2       3       4
```

Die Ausgabe zeigt, welche Ausgabekanäle offen sind. Jedes device besitzt dafür eine laufende Nummer, wobei das erste device die 2 erhält. Um zu erfahren, welcher der Ausgabekanäle aktiv ist, dient `dev.cur()` (*device current*).

```
> dev.cur()
windows
      4
```

Die Ausgabe besteht in der fortlaufenden Nummer des aktiven device. `dev.prev()` (*device previous*) und `dev.next()` geben bei mehreren geöffneten devices die Nummer desjenigen device zurück, das sich unmittelbar vor bzw. unmittelbar hinter dem aktiven device befindet. Diese Information kann etwa zum Wechseln des aktiven device mit `dev.set(Nummer)` verwendet werden, damit die Ausgabe der folgenden Grafikfunktionen dort erfolgt.

```
> dev.set(3)                                     # aktiviere device 3
windows
      3

> dev.set(dev.next())                           # aktiviere das folgende device
```

```
windows
 4
```

Das aktive device wird mit `dev.off()` geschlossen. Handelt es sich um ein Grafikfenster, hat dies denselben Effekt, wie das Fenster per Mausklick zu schließen. Die Ausgabe von `dev.off()` gibt an, welches fortan das aktive device ist. Ohne weitere geöffnete devices ist dies das *NULL-device* mit der Nummer 1. Alle offenen devices lassen sich gleichzeitig mit `graphics.off()` schließen.

```
> dev.off()                                # aktuelles device schließen
windows
 2

> graphics.off()                           # alle devices schließen
```

14.1.2 Grafiken speichern

Alles, was sich in einem Grafikfenster anzeigen lässt, kann auch als Datei gespeichert werden – in RStudio im Plots-Tab mit dem Eintrag **Export**. Ist ein Grafikfenster aktiviert, so ändert sich das Menü der R-Umgebung dahingehend, dass über **Datei: Speichern als:** die Grafik in vielen Formaten gespeichert werden kann. Als Alternative erlaubt ein sich durch Rechtsklick auf das Grafikfenster öffnendes Kontextmenü, die Grafik in wenigen Formaten zu speichern. Dasselbe Kontextmenü enthält auch Einträge, um die Grafik in einem bestimmten Format in die Zwischenablage zu kopieren und so direkt anderen Programmen verfügbar zu machen.

Grafiken lassen sich auch befehlsgesteuert ohne den Umweg eines Grafikfensters in Dateien speichern. Unabhängig davon, in welchem Format dies geschehen soll, sind dafür drei Arbeitsschritte notwendig: Zunächst muss die Datei als Ausgabekanal (also als aktives device) festgelegt werden. Dazu dient etwa `pdf()`, wenn die Grafik im PDF-Format zu speichern ist. Es folgen Befehle zum Erstellen von Diagrammen oder Einfügen von Grafikelementen, deren Ausgabe dann nicht auf dem Bildschirm erscheint, sondern direkt in die Datei umgeleitet wird. Schließlich ist der `dev.off()` oder `graphics.off()` Befehl notwendig, um die Ausgabe in die Datei zu beenden und das device zu schließen.

Als Dateiformate stehen viele der üblichen bereit, vgl. `?device` für eine Aufstellung. Beispielhaft seien hier `pdf()` und `jpeg()` betrachtet.²

```
> pdf(file="<Dateiname>", width=<Breite>, height=<Höhe>)
> jpeg(filename="<Dateiname>", width=<Breite>, height=<Höhe>,
+       units="px", quality=<Bildqualität>)
```

Unter `file` bzw. `filename` ist der Name der Ausgabedatei einzutragen – ggf. inkl. einer Pfadangabe (vgl. Abschn. 4.3). Sollen mehrere Grafiken mit gleichem Namensschema unter Zuhilfenahme einer fortlaufenden Nummer erzeugt werden, ist ein spezielles Namensformat zu verwenden, das in der Hilfe erläutert wird. Mit `width` und `height` wird die Größe der Grafik

²Alternativ stellt das Paket *Cairo* (Urbanek & Horner, 2014) die gleichnamige Funktion zur Verfügung, mit der Diagramme in vielen Dateiformaten auch in hoher Auflösung und mit automatischer Kantenglättung gespeichert werden können.

kontrolliert. Beide Angaben sind bei `pdf()` in der Einheit inch zu t tigen, w hrend bei `jpeg()`  ber das Argument `units` festgelegt werden kann, auf welche Ma einheit sie sich beziehen. Voreinstellung ist die Anzahl der pixel, als Alternativen stehen `in` (inch), `cm` und `mm` zur Auswahl. Schlie lich kann bei Bildern im JPEG-Format festgelegt werden, wie stark die Daten komprimiert werden sollen, wobei die Kompression mit einem Verlust an Bildinformationen verbunden ist. Das Argument `quality` erwartet einen sich auf die h chstm gliche Bildqualit t beziehenden Prozentwert – ein kleinerer Wert bedeutet hier eine geringere Bildqualit t, die dann st rkere Kompression f hrt daf r aber auch zu einer geringeren Dateigr  e.

Hier soll demonstriert werden, wie eine einfache Grafik im PDF-Format gespeichert wird.

```
> pdf("pdf_test.pdf")                                # device  ffnen (mit Dateinamen)
> plot(1:10, rnorm(10))                            # Grafik einzeichnen
> dev.off()                                         # device schlie en
```

Die Inhalte eines aktiven device k nnen durch `dev.copy()` in ein neues device kopiert werden. Dabei ist entweder die Nummer des bereits ge ffneten Ziel-Device zu nennen oder anzugeben, welche Art von device mit den bestehenden Inhalten neu ge ffnet werden soll.

```
> dev.copy(device=<Device-Typ>, ..., which=<Device-Nummer>)
```

Ist das Ziel-Device noch nicht ge ffnet, findet das `device` Argument Verwendung, das (ohne Anf hrungszeichen) den Namen einer Funktion erwartet, mit der ein device eines bestimmten Typs ge ffnet werden kann, etwa `pdf`. Andernfalls gibt `which` die Nummer des ge ffneten Ziel-Device an. Ben tigt das neu zu erstellende device seinerseits Argumente – etwa Dateinamen oder Angaben zur Gr  e der Grafik, so k nnen diese anstelle der ... durch Komma getrennt genannt werden.

```
> plot(1:10, rnorm(10))                            # Grafikfenster + Diagramm  ffnen

# Inhalt des Fensters in JPEG-Bild speichern,
# dabei Dateinamen und Kompressionsgrad angeben
> dev.copy(jpeg, filename="copied.jpg", quality=90)
> graphics.off()                                    # beide offenen devices schlie en
```

14.2 Streu- und Liniendiagramme

In zweidimensionalen Streudiagrammen (*scatterplots*) werden mit `plot()` Wertepaare in Form von Punkten in einem kartesischen Koordinatensystem dargestellt, wobei ein Wert die Position des Punkts entlang der Abszisse (*x*-Achse) und der andere Wert die Position des Punkts entlang der Ordinate (*y*-Achse) bestimmt. Die Punkte k nnen dabei f r ein Liniendiagramm durch Linien verbunden oder f r ein Streudiagramm als Punktfolge belassen werden.

14.2.1 Streudiagramme mit `plot()`

```
> plot(x=<Vektor>, y=<Vektor>, type="<Option>", main="<Diagrammtitel>",
+       sub="<Untertitel>", asp=<Seitenverh ltnis H ohe/Breite>)
```

Unter x und y sind die x bzw. y -Koordinaten der Punkte jeweils als Vektor einzutragen. Wird nur ein Vektor angegeben, werden seine Werte als y -Koordinaten interpretiert und die x -Koordinaten durch die Indizes des Vektors gebildet.³ Das Argument `type` hat mehrere mögliche Ausprägungen, die das Aussehen der Datenmarkierungen im Diagramm bestimmen (Tab. 14.1, Abb. 14.2). Der Diagrammtitel kann als Zeichenkette für `main` angegeben werden, der Untertitel für `sub`. Das Verhältnis von Höhe zu Breite jeweils einer Skaleneinheit im Diagramm kontrolliert das Argument `asp` (*aspect ratio*), für weitere Argumente vgl. `?plot.default`.

Die Koordinaten der Punkte können auch über andere Wege angegeben werden: Als Modellformel lautet der Aufruf `plot(<y-Koord.> ~ <x-Koord.>, data=<Datensatz>)`.⁴ Wenn die in der Modellformel verwendeten Variablen aus einem Datensatz stammen, ist dieser unter `data` zu nennen. Weiter ist es möglich, eine Liste mit zwei Komponenten `x` und `y` anzugeben, die die Koordinaten enthalten. Schließlich kann einfach eine Matrix mit zwei Spalten für die (x, y) -Koordinaten übergeben werden.

Tabelle 14.1: Mögliche Werte für das Argument `type` von `plot()`

Wert für <code>type</code>	Bedeutung
"p"	Punkte
"l"	durchgehende Linien. Durch eng gesetzte Stützstellen können Funktionskurven beliebiger Form approximiert werden
"b"	Punkte und Linien
"c"	unterbrochene Linien
"o"	Punkte und Linien, aber überlappend
"h"	senkrechte Linien zu jedem Datenpunkt (<i>spike plot</i>)
"s"	Stufendiagramm
"S"	Stufendiagramm mit anderer Reihenfolge von vertikalen und horizontalen Schritten zur nächsten Stufe
"n"	fügt dem Diagramm keine Datenpunkte hinzu (<i>no plotting</i>)

```
> vec <- rnorm(10)                                     # Daten

# das Argument xlab=NA entfernt die Bezeichnung der x-Achse
> plot(vec, type="p", xlab=NA, main="type p")
> plot(vec, type="l", xlab=NA, main="type l")
> plot(vec, type="b", xlab=NA, main="type b")
> plot(vec, type="o", xlab=NA, main="type o")
> plot(vec, type="s", xlab=NA, main="type s")
> plot(vec, type="h", xlab=NA, main="type h")
```

³Dagegen erzeugt `plot(<Faktor>, ...)` ein Säulendiagramm der Häufigkeiten jeder Faktorstufe (vgl. Abschn. 14.4), da `plot()` eine generische Funktion ist (vgl. Abschn. 15.2.6).

⁴Hat die Modellformel die Form `<y-Koord.> ~ <Faktor>`, werden in einem Diagramm getrennt für die von `<Faktor>` definierten Gruppen boxplots dargestellt (vgl. Abschn. 14.6.3).

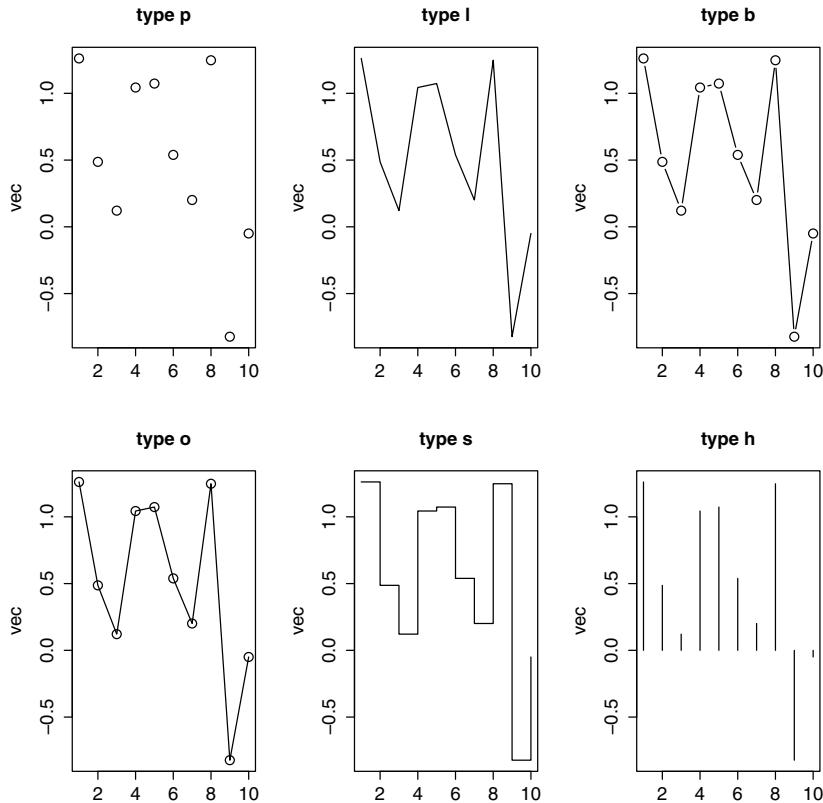


Abbildung 14.2: Mit `plot(type="<Option>")` erzeugbare Diagrammmarten

14.2.2 Datenpunkte eines Streudiagramms identifizieren

Werden viele Daten in einem Streudiagramm dargestellt, ist häufig nicht ersichtlich, zu welchem Wert ein bestimmter Datenpunkt gehört. Diese Information kann jedoch interessant sein, wenn etwa erst die grafische Betrachtung eines Datensatzes Besonderheiten der Verteilung verrät und die für bestimmte Datenpunkte verantwortlichen Untersuchungseinheiten identifiziert werden sollen. `identify()` erlaubt es, Werte in einem Streudiagramm interaktiv zu identifizieren.

```
> identify(x=(x-Koordinaten), y=(y-Koordinaten))
```

Für x und y sollten dieselben Daten in Form von Vektoren mit x- und y-Koordinaten übergeben werden, die zuvor in einem noch geöffneten Grafikfenster als Streudiagramm dargestellt wurden. Wird nur ein Vektor angegeben, werden seine Werte als y-Koordinaten interpretiert und die x-Koordinaten durch die Indizes des Vektors gebildet. Durch Ausführen von `identify()` ändert sich der Mauszeiger über der Diagrammfläche zu einem Kreuz. Mit einem Klick der linken

Maustaste wird derjenige Datenpunkt identifiziert, der der Mausposition am nächsten liegt und sein Index dem Diagramm neben dem Datenpunkt hinzugefügt. Die Konsole ist in dieser Zeit blockiert. Der Vorgang kann mehrfach wiederholt und schließlich durch Klicken der rechten Maustaste über ein Kontextmenü beendet werden, woraufhin `identify()` die Indizes der ausgemachten Punkte zurückgibt.

```
> vec <- rnorm(10)
> plot(vec)
> identify(vec)
```

14.2.3 Streudiagramme mit `matplot()`

So wie durch `plot()` ein Streudiagramm einer einzelnen Datenreihe erstellt wird, erzeugt `matplot()` ein Streudiagramm für mehrere Datenreihen gleichzeitig (Abb. 14.3).

```
> matplot(x=<Matrix>, y=<Matrix>, type="option", pch="Symbol")
```

Die Argumente sind dieselben wie für `plot()`, lediglich *x*- und *y*-Koordinaten können nun als Matrizen an *x* und *y* übergeben werden, wobei jede ihrer Spalten als eine separate Datenreihe interpretiert wird. Haben dabei alle Datenreihen dieselben *x*-Koordinaten, kann *x* auch ein Vektor sein. Wird nur eine Matrix mit Koordinaten angegeben, werden diese als *y*-Koordinaten gedeutet und die *x*-Koordinaten durch die Zeilenindizes der Werte gebildet. In der Voreinstellung werden die Datenreihen in unterschiedlichen Farben dargestellt. Als Symbol für jeden Datenpunkt dienen die Ziffern 1–9, die mit der zur Datenreihe gehörenden Spaltennummer korrespondieren. Mit dem Argument `pch` können auch andere Symbole Verwendung finden (vgl. Abschn. 14.3.1, Abb. 14.5).

```
> vec <- seq(from=-2*pi, to=2*pi, length.out=50)
> mat <- cbind(2*sin(vec), sin(vec-(pi/4)), 0.5*sin(vec-(pi/2)))
> matplot(vec, mat, type="b", xlab=NA, ylab=NA, pch=1:3,
+           main="Sinuskurven")
```

14.3 Diagramme formatieren

`plot()` akzeptiert wie auch andere High-Level-Grafikfunktionen eine Vielzahl weiterer Argumente, mit denen ein Diagramm flexibel angepasst werden kann. Einige der wichtigsten Möglichkeiten zur individuellen Gestaltung werden im folgenden vorgestellt.

14.3.1 Grafikelemente formatieren

Die Formatierung von Grafikelementen ist in vielen Aspekten variabel, etwa hinsichtlich der Art, Größe und Farbe der verwendeten Symbole oder der Orientierung der Achsenbeschriftungen. Zu diesem Zweck akzeptieren die meisten High-Level-Funktionen einen gemeinsamen Satz zusätzlicher Argumente, auch wenn diese nicht immer in den jeweiligen Hilfe-Seiten mit aufgeführt sind. Die gebräuchlichsten von ihnen sind in Tab. 14.2 beschrieben.

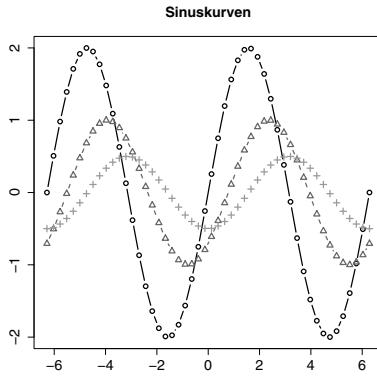


Abbildung 14.3: Mit `matplot()` erzeugtes Streudiagramm

Tabelle 14.2: Diagrammoptionen, die in `par()` und high-level Grafik-Funktionen gesetzt werden können

Argument	Wert	Bedeutung
bty	"o", "n"	Bei "o" wird ein Rahmen um die die Datenpunkte enthaltende Plot-Region gezogen, bei "n" nicht
cex	$\langle \text{Zahl} \rangle$	Vergrößerungsfaktor für die Datenpunkt-Symbole. Voreinstellung ist der Wert 1
cex.axis	$\langle \text{Zahl} \rangle$	Vergrößerungsfaktor für die Achsenbeschriftungen. Voreinstellung ist der Wert 1
cex.lab	$\langle \text{Zahl} \rangle$	Vergrößerungsfaktor für die Schrift der Achsenbezeichnungen. Voreinstellung ist der Wert 1
col	" $\langle \text{Farbe} \rangle$ "	Farbe der Datenpunkt-Symbole sowie bei <code>par()</code> zusätzlich des Rahmens um die Plot-Region (vgl. Abschn. 14.3.2 für mögliche Werte)
las	0, 1, 2, 3	Orientierung der Achsenbeschriftungen. Für senkrecht zur Achse stehende Beschriftungen ist der Wert auf 2 zu setzen
lty	1, 2, 3, 4, 5, 6 bzw. $\langle \text{Schlüsselwort} \rangle$	Linientyp: Schlüsselwörter sind "solid", "dashed", "dotted", "dotdash", "longdash", "twodash" (Abb. 14.5)
lwd	$\langle \text{Zahl} \rangle$	Linienstärke, auch bei Datenpunktssymbolen. Voreinstellung ist der Wert 1.
pch	$\langle \text{Zahl } 1\text{--}25 \rangle$ bzw. " $\langle \text{Buchstabe} \rangle$ "	Art der Datenpunkt-Symbole. Dabei steht etwa 16 für den ausgefüllten Punkt. Für andere Werte vgl. <code>?points</code> und Abb. 14.5. Wird ein Buchstabe angegeben, dient dieser als Symbol der Datenpunkte

Tabelle 14.2: (Forts.)

xpd	NA, TRUE, FALSE	Grafikelemente können nur in der Plot-Region eingefügt werden (Voreinstellung FALSE: clipping). TRUE: gesamte Figure-Region steht zur Verfügung, NA: gesamte Device-Region (vgl. Abschn. 14.1.1, 14.5.2)
-----	-----------------	--

Anstatt die in Tab. 14.2 genannten Argumente direkt beim Aufruf von Grafikfunktionen mit anzugeben, können sie durch die separate Funktion `par(<Option>=<Wert>)` festgelegt werden. `par()` kann darüber hinaus noch weitere Einstellungen ändern, die in Tab. 14.3 aufgeführt sind. Die aktuell für das aktive device gültigen Einstellungen lassen sich durch `par("Opt1", "Opt2", ...)` als Liste ausgeben, d. h. durch Nennung der relevanten Argumente ohne Zuweisung von Werten. Ohne weitere Argumente gibt `par()` die aktuellen Werte für alle veränderbaren Parameter aus.

Tabelle 14.3: Grafikoptionen, die nur über `par()` verändert werden können

Argument	Wert	Bedeutung
mar	<code><Vektor></code>	Ränder zwischen Plot- und Figure-Region eines Diagramms (Abb. 14.1). Angabe als Vielfaches der Zeilenhöhe in Form eines Vektors mit vier Elementen, die jeweils dem unteren, linken, oberen und rechten Rand entsprechen. Voreinstellung ist <code>c(5, 4, 4, 2)</code>
mai	<code><Vektor></code>	Wie <code>mar</code> , jedoch in der Einheit inch
oma	<code><Vektor></code>	Ränder zwischen Figure- und Device-Region einer Grafik (Abb. 14.1). Bei aufgeteilten Diagrammen zwischen den zusammengefassten Figure-Regionen und der Device-Region. Angabe wie bei <code>mar</code> . Voreinstellung ist <code>c(0, 0, 0, 0)</code> , d. h. die Figure-Region füllt die Device-Region vollständig aus
omi	<code><Vektor></code>	Wie <code>oma</code> , jedoch in der Einheit inch

Die mit `par()` geänderten Parameter sind Einstellungen für das aktive device. Sie gelten für alle folgenden Ausgaben in dieses device bis zur nächsten expliziten Änderung, oder bis ein neues device aktiviert wird. Der auf der Konsole nicht sichtbare Rückgabewert von `par()` enthält die alten Einstellungen der geänderten Optionen in Form einer Liste, die auch direkt wieder an `par()` übergeben werden kann. Auf diese Weise lassen sich Einstellungen temporär ändern und dann wieder auf den Ursprungswert zurücksetzen (Abb. 14.4).

```
# Werte ändern und alte speichern
> op <- par(col="gray60", lwd=2, pch=16)
> plot(rnorm(10), main="Grau, fett, gefüllte Kreise")

# Parameter auf ursprüngliche Werte zurücksetzen
> par(op)
> plot(rnorm(10), main="Standardformatierung")
```

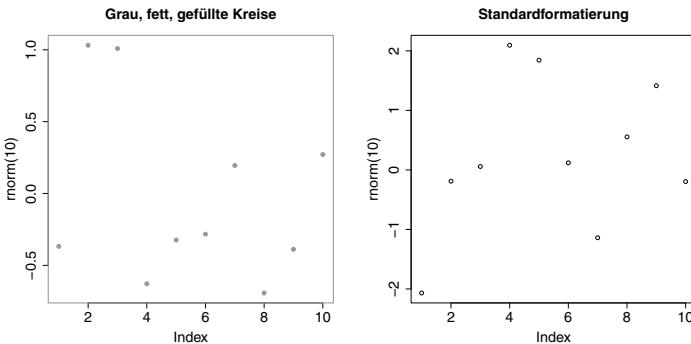


Abbildung 14.4: Verwendung von `par()` zur Diagrammformatierung

Abbildung 14.5 veranschaulicht die mit `lty` und `pch` einstellbaren Linientypen und Datenpunkt-Symbole (vgl. Abschn. 14.5 für das Einfügen von Elementen in ein Diagramm). Symbole 21–25 sind ausgefüllte Datenpunkte, deren Füllfarbe in Zeichenfunktionen über das Argument `bg="<Farbe>"` definiert wird, während `col="<Farbe>"` die Farbe des Randes bezeichnet (vgl. Abschn. 14.3.2).

```
# Koordinaten der Datenpunkte
> X <- row(matrix(numeric(6*11), nrow=6, ncol=11))
> Y <- col(matrix(numeric(6*11), nrow=6, ncol=11))

# leeres Diagramm mit der richtigen Skalierung
> plot(0:6, seq(1, 11, length.out=7), type="n", axes=FALSE,
+       main="pch Datenpunkt-Symbole und lty Linientypen")

# Symbole für pch mit grauem Hintergrund für 21-25
> points(X[1:26], Y[1:26], pch=0:25, bg="gray", cex=2)

# Linien für lty
> matlines(X[, 6:11], Y[, 6:11], lty=6:1, lwd=2, col="black")

# erklärender Text - Nummer der pch Symbole und lty Linientypen
> text(X[1:26]-0.3, Y[1:26], labels=0:25)
> text(rep(0.7, 6), Y[1, 6:11], labels=6:1)
> text(0, 9, labels="Linientypen für lty", srt=90, cex=1.2)
> text(0, 3, labels="Symbole für pch", srt=90, cex=1.2)
```

14.3.2 Farben spezifizieren

Häufig ist es sinnvoll, Diagrammelemente farblich hervorzuheben, etwa um die Zusammengehörigkeit von Punkten innerhalb von Datenreihen zu kennzeichnen und verschiedene Datenreihen leichter voneinander unterscheidbar zu machen. Auch Text- und Hintergrundfarben können

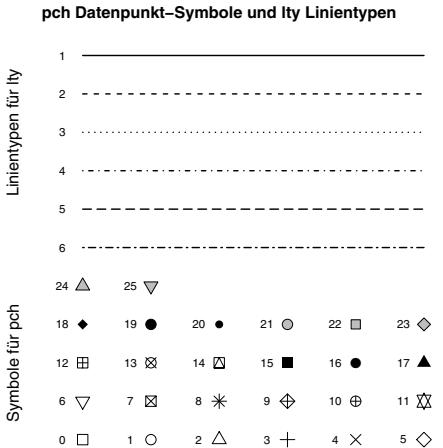


Abbildung 14.5: Datenpunkt-Symbole und Linientypen zur Verwendung für die Argumente `pch` und `lty` von Grafikfunktionen

in Diagrammen frei gewählt werden. Zu diesem Zweck lassen sich Farben in unterschiedlicher Form an die entsprechenden Funktionsargumente (meist `col`) übergeben:

- Als Farbname, z. B. "green" oder "blue", vgl. `colors()` für mögliche Werte.
- Als natürliche Zahl, die als Index für die derzeit aktive Farbpalette interpretiert wird. Eine Farbpalette ist dabei ein vordefinierter Vektor von Farben, der mit `palette()` ausgegeben werden kann. Die voreingestellte Palette beginnt mit den Farben "black", "red", "green3", "blue" – der Index 2 entspräche also der Farbe Rot. Die Palette kann gewechselt werden, indem an `palette(Vektor)` ein Vektor mit Farbnamen übergeben wird. Der (auf der Konsole unsichtbare) Rückgabewert enthält die ersetzte Palette und kann für einen temporären Wechsel der Palette in einem Vektor zwischengespeichert und später wieder an `palette()` übergeben werden. Alternativ stellt `palette("default")` die ursprünglich voreingestellte Palette wieder her. Für eine Beschreibung der verfügbaren Paletten vgl. `?rainbow`. Die dort genannten Funktionen können etwa dazu eingesetzt werden, die aktive Farbpalette zu ändern, indem ihre Ausgabe an `palette()` übergeben wird, z. B. mit `palette(rainbow(10))`.⁵
- Im Hexadezimalformat, wobei die Intensitäten der Monitor-Grundfarben Rot, Grün und Blau in der Form "#RRGGBB" mit Werten für RR, GG und BB im Bereich von 00 bis FF angegeben werden. "#FF0000" entspräche Rot, "#00FF00" Grün.

`col2rgb(<Farbe>)` wandelt Farbnamen, Palettenindizes und Farben im Hexadezimalformat in einen Spaltenvektor um, der die Intensitäten der Monitor-Grundfarben Rot, Grün und Blau im

⁵Das Paket `RColorBrewer` (Neuwirth, 2011) definiert eine Reihe von Paletten, deren Farben für besonders gute Unterscheidbarkeit unter verschiedenen Randbedingungen optimiert wurden – etwa für Farbfehlsehige oder Graustufen-Ausdrucke.

Wertebereich von 0–255 enthält. Namen und Hexadezimalzahlen müssen dabei in Anführungszeichen gesetzt werden. Da die Spezifizierung von Farben im Hexadezimalformat nicht sehr intuitiv ist, stellt R verschiedene Funktionen bereit, mit denen Farben auf einfachere Art definiert werden können. Diese Funktionen geben dann die bezeichnete Farbe im Hexadezimalformat aus.

- Mit `rgb(red=<Rot>, green=<Grün>, blue=<Blau>)` können die Intensitäten der Monitor-Grundfarben Rot, Grün und Blau mit Zahlen im Wertebereich von 0–1 angegeben werden. Andere Höchstwerte lassen sich über das Argument `maxColorValue` festlegen.⁶ So gibt etwa `rgb(0, 1, 1)` die Farbe "#00FFFF" (Cyan) aus. Bei von `col2rgb()` erzeugten Vektoren ist `rgb()` mit dem Argument `maxColorValue=255` zu verwenden.
- Analog erzeugt `hsv(h=<Farbton>, s=<Sättigung>, v=<Helligkeit>)` (*hue, saturation, value*) Farben, die mit Zahlen im Wertebereich von 0–1 für Farbton, Sättigung und Helligkeit definiert werden.⁷ So entspricht etwa `hsv(0.1666, 1, 1)` der Farbe "#FFFF00" (Gelb). `rgb2hsv(r=<Rot>, g=<Grün>, b=<Blau>)` rechnet von RGB-Farben in HSV-Werte um.
- `hcl(h=<Farbton>, c=<Sättigung>, l=<Helligkeit>)` (*hue, chroma, luminance*) erzeugt Farben im CIE Luv Koordinatensystem, das auf gleiche perzeptuelle Unterschiedlichkeit von im Farbraum gleich weit entfernten Farben abzielt. Dabei ist `h` ein Winkel im Farbkreis im Bereich von 0–360°, `c` die Sättigung, deren Höchstwert vom Farbton und der Luminanz abhängt und schließlich `l` die Luminanz im Bereich von 0–100.
- `gray(<Grauwert>)` akzeptiert eine Zahl im Wertebereich von 0–1, die als Helligkeit einer achromatischen Farbe mit identischen RGB-Werten interpretiert wird. So erzeugt etwa `gray(0.5)` die graue Farbe "#808080". Das optionale Argument `alpha` kontrolliert den Grad simulierter Transparenz (vgl. Fußnote 6).
- `colorRamp(<Farben>)` und `colorRampPalette(<Farben>)` erstellen einen Farbverlauf, indem sie im Farbraum gleichmäßig zwischen den übergebenen Farben interpolieren. Über das Argument `alpha=TRUE` gilt dies auch für den Grad simulierter Transparenz. Das Ergebnis von `colorRamp()` ist dabei eine Funktion, die ihrerseits eine Zahl im Bereich [0, 1] akzeptiert und als relative Distanz zwischen der ersten und letzten zu interpolierenden Farbe interpretiert. Dagegen ist die Ausgabe von `colorRampPalette()` eine Funktion, die analog zu den vordefinierten Palettenfunktionen wie `rainbow()` oder `cm.colors()` arbeitet.
- Eine gegebene Farbe kann mit `adjustcolor()` hinsichtlich verschiedener Attribute kontrolliert verändert werden – etwa der Helligkeit, der Sättigung oder der simulierten Durchsichtigkeit.

⁶Ein vierter Wert zwischen 0 und 1 kann den Grad des *alpha-blendings* für simulierte Transparenz definieren. Niedrige Werte stehen für sehr durchlässige, hohe Werte für opaque Farben (Abb. 12.2). Diese Art von Transparenz wird nur von manchen devices unterstützt, etwa von `pdf()` oder `png()`.

⁷Für weitere Funktionen zur Verwendung verschiedener Farbräume vgl. `?convertColor` sowie das Paket `colorspace` (Ihaka, Murrell, Fisher & Zeileis, 2013).

14.3.3 Achsen formatieren

- Ob durch die Darstellung von Datenpunkten in einem Diagramm automatisch auch Achsen generiert werden, kontrollieren in High-Level-Funktionen die Argumente `xaxt` für die `x`-Achse, `yaxt` für die `y`-Achse und `axes` für beide Achsen gleichzeitig. Während für `xaxt` und `yaxt` der Wert "`n`" übergeben werden muss, um die Ausgabe der entsprechenden Achse zu unterdrücken, akzeptiert das Argument `axes` dafür den Wert `FALSE`.
- Achsen können mit `axis()` auch separat einem Diagramm hinzugefügt werden, wobei sich Lage, Beschriftung und Formatierung der Achsenmarkierungen festlegen lassen (vgl. Abschn. 14.5.8).
- Die Argumente `xlim=⟨Vektor⟩` und `ylim=⟨Vektor⟩` von High-Level-Funktionen legen den durch die Achsen abgedeckten Wertebereich in Form eines Vektors mit dem kleinsten und größten Achsenwert fest. Fehlen diese Argumente, wird jeder Bereich automatisch anhand der darzustellenden Daten bestimmt.
- Bei expliziten Angaben für `xlim` oder `ylim` gehen die Achsen auf beiden Seiten um 4% über den angegebenen Wertebereich hinaus. Um dies zu verhindern, sind in High-Level-Funktionen die Argumente `xaxs="i"` und `yaxs="i"` zu setzen.
- Achsenbezeichnungen können in High-Level-Funktionen über die Argumente `xlab="⟨Name⟩"` und `ylab="⟨Name⟩"` gewählt oder mit Setzen auf `NA` unterdrückt werden.
- Eine logarithmische Skalierung zur Basis 10 lässt sich in High-Level-Funktionen getrennt für jede Achse mit dem Argument `log="x"`, `log="y"` bzw. für beide Achsen gemeinsam mit `log="xy"` erzeugen.
- Die Orientierung der Achsenmarkierungen legt das Argument `las` von `par()` fest. Senkrecht ausgerichtete Beschriftungen beanspruchen dabei häufig mehr vertikalen Platz, als es die Randeinstellungen vorsehen. Mit dem Argument `mar` von `par()` lässt sich der Rand zwischen Plot- und Figure-Region entsprechend anpassen.

14.4 Säulen- und Punktdiagramme

Mit `barplot()` erstellte Säulendiagramme eignen sich zur Darstellung von Kennwerten von Variablen, die getrennt für verschiedene Gruppen berechnet wurden. Dazu zählen etwa absolute oder relative Häufigkeiten von Gruppenzugehörigkeiten oder der jeweilige Mittelwert einer Variable in verschiedenen Stichproben. Der Kennwert jeder Gruppe wird dabei durch eine Säule repräsentiert, deren Höhe seine Größe widerspiegelt.⁸

⁸Für grafisch aufwendiger gestaltete Säulendiagramme vgl. `barp()` aus dem Paket `plotrix` (Lemon, 2006).

14.4.1 Einfache Säulendiagramme

Sollen Kennwerte einer Variable getrennt für Gruppen dargestellt werden, die sich aus den Stufen eines einzelnen Faktors ergeben, lautet die Grundform von `barplot()`:

```
> barplot(height=<Vektor>, <Argumente>)
```

- Unter `height` ist ein Vektor einzutragen, wobei jedes seiner Elemente den Kennwert für jeweils eine Bedingung repräsentiert und damit die Höhe einer Säule festlegt.
- `horiz` bestimmt, ob vertikale Säulen (Voreinstellung `FALSE`) oder horizontale Balken gezeichnet werden. Der (auf der Konsole nicht sichtbare) Rückgabewert von `barplot()` enthält die x -Koordinaten der eingezeichneten Säulen bzw. die y -Koordinaten der Balken.
- `space` legt den Abstand zwischen den Säulen fest, in der Voreinstellung beträgt er 0.2.
- `names.arg` nimmt einen Vektor von Zeichenketten an, der die Beschriftung der Säulen definiert.
- Ist das Minimum des mit `ylim` definierten Wertebereichs der y -Achse größer als 0, muss `xpd=FALSE` gesetzt werden, um Säulen nicht unterhalb der x -Achse zeichnen zu lassen. Andernfalls erscheinen in der Voreinstellung `xpd=TRUE` die Säulen unterhalb der x -Achse, auch wenn diese nicht bei 0 beginnt.

Als Beispiel diene das Ergebnis mehrerer simulierter Würfe eines sechsseitigen Würfels, wobei einmal die absoluten und einmal die relativen Häufigkeiten dargestellt werden sollen (Abb. 14.6).

```
> dice <- sample(1:6, 100, replace=TRUE)           # Würfelwürfe
> (dTab <- xtabs(~ dice))                        # abs. Häufigkeiten
dice
 1   2   3   4   5   6
17  23  14  20  14  12

> barplot(dTab, ylim=c(0, 30), xlab="Augenzahl", ylab="N", col="black",
+           main="Absolute Häufigkeiten")

> barplot(prop.table(dTab), ylim=c(0, 0.3), xlab="Augenzahl",
+           ylab="relative Häufigkeit", col="gray50",
+           main="Relative Häufigkeiten")
```

14.4.2 Gruppierte und gestapelte Säulendiagramme

Gruppierte Säulendiagramme stellen Kennwerte von Variablen getrennt für Gruppen dar, die sich aus der Kombination zweier Faktoren ergeben. Zu diesem Zweck kann die Zusammengehörigkeit einer aus mehreren Säulen bestehenden Gruppe grafisch durch ihre räumliche Nähe innerhalb der Gruppe und die gleichzeitig größere Distanz zu anderen Säulengruppen kenntlich gemacht werden. Eine weitere Möglichkeit besteht darin, jede Einzelsäule nicht homogen, sondern als Stapel mehrerer Segmente darzustellen (Abb. 14.7).

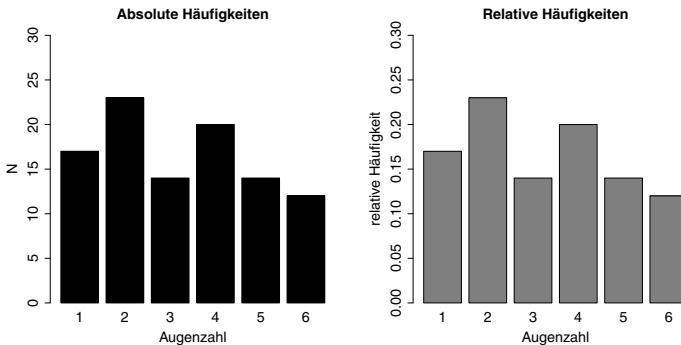


Abbildung 14.6: Säulendiagramme

- Für gruppierte oder gestapelte Säulendiagramme werden die Daten an `height` in Form einer Matrix übergeben, deren Werte die Säulenhöhen bzw. Balkenlängen festlegen.
- Das Argument `beside` kontrolliert, welche Darstellungsart gewählt wird: auf `TRUE` gesetzt bewirkt es Säulengruppen, in der Voreinstellung `FALSE` gestapelte Säulen.
- Ist `beside=FALSE`, definiert jede Spalte der Datenmatrix die innere Zusammensetzung einer Säule, indem die einzelnen Werte einer Spalte die Höhe der Segmente bestimmen, aus denen die Säule besteht. Bei `beside=TRUE` definiert eine Spalte der Datenmatrix eine Säulengruppe, deren jeweilige Höhen durch die Einzelwerte in der Spalte festgelegt sind.
- Bei gruppierten Säulendiagrammen muss `space` mit einem Vektor definiert werden. Das erste Element stellt den Abstand innerhalb der Gruppen dar, das zweite jenen zwischen den Gruppen. Voreinstellung ist der Vektor `c(0, 1)`.
- `names.arg` nimmt einen Vektor von Zeichenketten an, der die Beschriftung der Säulengruppen definiert.
- `legend.text` kontrolliert, ob eine Legende eingefügt wird, Voreinstellung ist `FALSE`. Auf `TRUE` gesetzt erscheint eine Legende, die auf den Zeilennamen der Datenmatrix basiert und sich auf die Bedeutung der Säulen innerhalb einer Gruppe bzw. auf die Segmente einer Säule bezieht. Alternativ können die Einträge der Legende als Vektor von Zeichenketten angegeben werden.

```

> roll11  <- dice[1:50]                      # erste Serie von Würfelwürfen
> roll12  <- dice[51:100]                     # zweite Serie
> rollAll <- rbind(table(roll11), table(roll12))    # Matrix Gesamtdaten
> rownames(rollAll) <- c("first", "second")
> rollAll
      1   2   3   4   5   6
first  5  15   9   8   7   6
second 12   8   5  12   7   6

```

Die in jeder der sechs Gruppen vorhandenen zwei Säulen sollten farblich getrennt werden, um die Zugehörigkeit zur ersten bzw. zweiten Substichprobe deutlich zu machen. Dazu wird an `col` ein Vektor mit zwei Farbnamen übergeben, den R intern so häufig recycled, wie es Säulengruppen gibt.

```
> barplot(rollAll, beside=FALSE, legend.text=TRUE, xlab="Augenzahl",
+         ylab="N", main="Abs. Häufigkeiten in zwei Substichproben")

> barplot(rollAll, beside=TRUE, ylim=c(0, 15), col=c("red", "green"),
+         legend.text=TRUE, xlab="Augenzahl", ylab="N",
+         main="Abs. Häufigkeiten in zwei Substichproben")
```

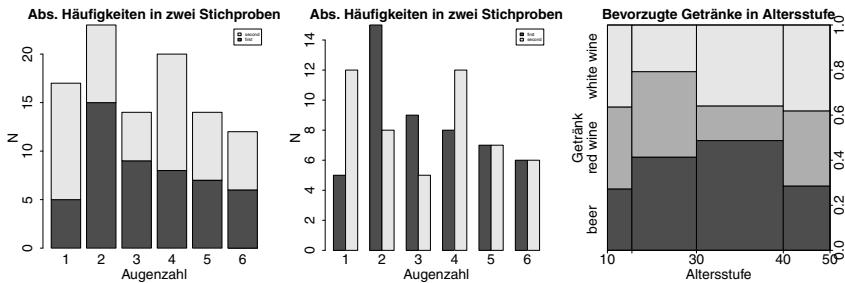


Abbildung 14.7: Gestapeltes und gruppiertes Säulendiagramm absoluter sowie spineplot bedingter relativer Häufigkeiten

Eine Verallgemeinerung gestapelter Säulendiagramme, mit denen die bedingten relativen Häufigkeiten einer kategorialen Variable in Abhängigkeit von einer anderen Variable dargestellt werden können, erzeugt `spineplot()` (Abb. 14.7).⁹

```
> spineplot(x=<Vektor>, y=<Faktor>, breaks=<Grenzen>,
+            xaxlabels=<Bezeichnungen>, yaxlabels=<Bezeichnungen>)"
```

Die Daten in `x` und `y` gelten als Werte, die an denselben Beobachtungsobjekten erhoben wurden. Im Fall von `x` sind dies entweder Ausprägungen einer kategorialen (`x` ist ein Faktor) oder einer quantitativen Variable (`x` ist ein numerischer Vektor), während `y` ein Faktor sein muss. Das Diagramm setzt sich aus nahtlos nebeneinander stehenden Säulen zusammen, deren Anzahl sowie Breite durch `x` und deren Aufteilung in Segmente durch `y` definiert ist: Sofern `x` ein Faktor ist, stellt das Diagramm für jede seiner Ausprägungen eine Säule dar, die aus so vielen Segmenten besteht, wie `y` Ausprägungen hat. Der Flächeninhalt eines Segments spiegelt die relative Häufigkeit der zugehörigen Stufenkombination in der Gesamtstichprobe wider. Die bedingten relativen Häufigkeiten der Stufen von `y` in der Stufe von `x` werden dabei über die Höhe der Segmente visualisiert. Die Breite der Säulen repräsentiert die relativen Häufigkeiten der Stufen von `x`, so dass insgesamt die Verteilungen beider Variablen im Diagramm abzulesen sind.

⁹Für die Verteilung eines dichotomen Merkmals in Abhängigkeit einer kontinuierlichen Variable vgl. `cdplot()` (Abb. 8.1). Vergleiche `mosaicplot()` (Abb. 8.5) für die gemeinsame Verteilung von mehr als zwei kategorialen Variablen.

Ist **x** eine quantitative Variable, wird ihr Wertebereich zunächst entweder automatisch in disjunkte Intervalle eingeteilt, oder anhand eines für **breaks** zu übergebenden Vektors mit den Intervallgrenzen. Alternativ kann für **x** und **y** zum einen eine Modellformel der Form $\langle y \rangle \sim \langle x \rangle$ genannt werden. Zum anderen lassen sich die Daten in Gestalt einer zweidimensionalen Kreuztabelle der gemeinsamen Häufigkeiten von **x** und **y** übergeben – diese Kreuztabelle ist gleichzeitig der auf der Konsole nicht sichtbare Rückgabewert von **spineplot()**. Die Bezeichnungen für die durch **x** und **y** definierten Bedingungen ergeben sich aus deren Faktorstufen, können aber auch explizit durch Vektoren aus Zeichenketten für **xaxlabels** und **yaxlabels** genannt werden.

Die folgenden Daten sollen das Ergebnis einer Umfrage simulieren, in der Personen unterschiedlichen Alters ihre Präferenz für ein alkoholisches Getränk abgeben.

```
> N      <- 100                      # Anzahl Personen
> age    <- sample(18:45, N, replace=TRUE)      # Alter
> drinks <- c("beer", "red wine", "white wine")  # mögliche Getränke
> pref   <- factor(sample(drinks, N, replace=TRUE)) # Präferenzen

# Intervallgrenzen zur Diskretisierung von age
> xRange <- round(range(age), -1) + c(-10, 10)
> lims   <- seq(xRange[1], xRange[2], by=10)
> spineplot(x=age, y=pref, xlab="Altersstufe", ylab="Getränk",
+            breaks=lims, main="Bevorzugte Getränke pro Altersstufe")
```

14.4.3 Dotchart

dotchart() dient der Darstellung von Rohdaten einzelner Beobachtungsobjekte, aber auch von aggregierten Kennwerten von Variablen. Dies können etwa absolute oder relative Häufigkeiten von Gruppenzugehörigkeiten oder die jeweiligen Mittelwerte einer Variable in verschiedenen Stichproben sein. Jeder Wert wird dabei durch einen Punkt repräsentiert, dessen *x*-Koordinate seine Größe widerspiegelt und dessen *y*-Koordinate das Beobachtungsobjekt codiert. Das Ergebnis von **dotchart()** ist analog zu einem horizontalen Balkendiagramm, wobei statt der Balken lediglich Punkte eingezeichnet werden.

```
> dotchart(x=<Daten>, labels=<'Namen'>, groups=<Faktor>, gdata=<Daten>)
```

Für **x** ist der Datenvektor anzugeben. Über das Argument **labels** lassen sich mittels eines Vektors aus Zeichenketten derselben Länge wie **x** die Bezeichnungen der Datenpunkte nennen. Sollen Daten aus verschiedenen, durch die Kombination zweier Faktoren gebildeten Gruppen dargestellt werden, sind die Daten in Form einer Matrix zu übergeben, deren Werte die *x*-Koordinaten der Punkte festlegen. Dabei definiert jede Spalte der Datenmatrix eine Punktgruppe, deren Punkte vertikal nahe beieinander gezeichnet werden, wohingegen die durch verschiedene Spalten definierten Punkte stärker räumlich getrennt sind.

Stellen die Daten Kennwerte verschiedener Gruppen dar, können sie auch als Vektor **x** unter gleichzeitiger Angabe von **groups** übergeben werden. Für **groups** ist dann ein Faktor derselben Länge wie **x** zu nennen, der die Gruppenzugehörigkeit jedes Wertes definiert – auf diese Weise lassen sich auch Daten ungleich großer Gruppen darstellen. Zusätzlich zu den in **x** enthaltenen Werten lassen sich für **gdata** weitere Daten in Form eines Vektors mit so vielen Elementen

angeben, wie Gruppen vorhanden sind. Jeder Wert von `gdata` wird als zu jeweils einer Gruppe gehörig interpretiert und als einzelner Vergleichswert eingezeichnet. Diese Möglichkeit ist z. B. geeignet, um neben den Rohdaten einer Gruppe gleichzeitig auch einen aggregierten Kennwert der Daten mit darzustellen (Abb. 14.8).

```
> Nj <- 5                                     # Gruppengröße
> DV1 <- rnorm(Nj, 20, 2)                     # Daten Gruppe 1
> DV2 <- rnorm(Nj, 25, 2)                     # Daten Gruppe 2
> DV <- c(DV1, DV2)                          # Gesamt-Daten
> IV <- gl(2, Nj)                            # Gruppenzugehörigkeit
> Mj <- tapply(DV, IV, mean)                 # Gruppenmittel
> dotchart(DV, gdata=Mj, color=rep(c("red", "blue", "green"), each=Nj),
+           gcolor="black", groups=IV, labels=rep(LETTERS[1:Nj], 3),
+           xlab="Messwerte", ylab="Gruppen", pch=20,
+           main="Individuelle Ergebnisse und Mittel aus 2 Gruppen")
```

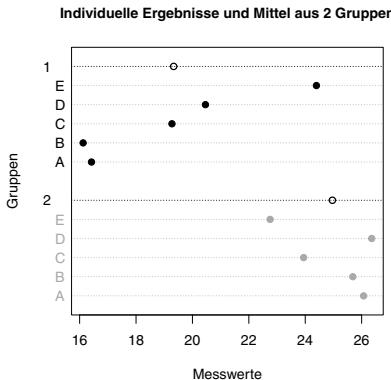


Abbildung 14.8: Dotchart von individuellen Messwerten in zwei Gruppen samt zugehöriger Mittelwerte

14.5 Elemente einem bestehenden Diagramm hinzufügen

Ein bereits erstelltes Diagramm lässt sich nachträglich durch zusätzliche Elemente erweitern, die durch Low-Level-Funktionen erzeugt werden (Tab. 14.4). Auch einige High-Level-Funktionen besitzen das Argument `add=TRUE`, durch das ihre Ausgabe dem derzeit aktiven device hinzugefügt wird, ohne dessen Inhalt zu löschen. In diesem Fall kann meist durch das Argument `at=<Position>` bestimmt werden, an welcher Stelle der Plot-Region die zusätzlichen Daten erscheinen sollen. Ist kein `add` Argument vorhanden, bewirkt die vorhergehende Ausführung von `par(new=TRUE)`, dass das Ergebnis des folgenden High-Level-Befehls im schon geöffneten device erscheint. Später eingefügte Elemente werden immer über bereits bestehende gezeichnet – neue Inhalte übermalen also ältere Inhalte, die sich an derselben Stelle befinden.

Tabelle 14.4: Mögliche Diagrammelemente und die sie hinzufügenden Funktionen

Diagrammelement	hinzufügende Low-Level-Grafikfunktion
Punkte	<code>points()</code> , <code>matpoints()</code>
Geradenabschnitte	<code>lines()</code> , <code>matlines()</code> , <code>segments()</code> , <code>abline()</code>
Gitter, Pfeile	<code>grid()</code> , <code>arrows()</code>
Polygone	<code>box()</code> , <code>rect()</code> , <code>polygon()</code>
interpolierte Punkte	<code>approx()</code> , <code>spline()</code> , <code>smooth.spline()</code> , <code>xspline()</code>
Funktionsgraphen	<code>curve()</code>
Text	<code>title()</code> , <code>legend()</code> , <code>text()</code> , <code>mtext()</code>
Achsen	<code>axis()</code>

Die Verwendung von Low-Level-Funktionen setzt voraus, dass bereits ein Diagramm mit einer High-Level-Funktion erstellt wurde. Um diesen Schritt zu überspringen und ein Diagramm ausschließlich aus Low-Level-Funktionen aufzubauen, kann aber auch ein device mit `dev.new(); plot.new()` geöffnet und zum Einfügen von Elementen vorbereitet werden.

14.5.1 Koordinaten in einem Diagramm identifizieren

Die meisten nachträglich einzufügenden Diagrammelemente erfordern für ihre Positionierung die Angabe von (x, y) -Koordinaten in einem Koordinatensystem, das durch die in der Plot-Region (vgl. Abschn. 14.1.1, Abb. 14.1) bereits dargestellten Daten festgelegt ist. Diese *User-Koordinaten* neu einzufügender Elemente ergeben sich oft direkt aus den Daten, mitunter soll ein Element aber auch frei plaziert werden. Eine Alternative zur Bestimmung der dafür geeigneten Koordinaten durch Versuch und Irrtum bietet `locator(n=<Anzahl>)`, nachdem ein Diagrammfenster erstellt wurde und noch geöffnet ist.

Das Argument `n` legt fest, wie viele Koordinaten zu bestimmen sind. Durch den Aufruf von `locator()` ändert sich der Mauszeiger zu einem Kreuz, sobald er sich über der Diagrammfläche befindet. Mit einem Klick der linken Maustaste werden die Koordinaten der Mausposition zwischengespeichert, währenddessen ist die Konsole für Eingaben blockiert. Der Vorgang kann mehrfach wiederholt werden und endet entweder, nachdem `n` Punkte gespeichert wurden, oder über ein sich durch Klicken der rechten Maustaste öffnendes Kontextmenü. Daraufhin gibt `locator()` eine Liste mit den Komponenten `x` und `y` zurück, die in Form jeweils eines Vektors die (x, y) -Koordinaten der angeklickten Punkte enthalten.

```
> plot(rnorm(10))                      # Streudiagramm
> (xy <- locator(n=3))                 # 3 Punkte durch Klicken auswählen
$x
[1] 2.402735 7.008204 3.141016

$y
[1] 0.656082382 -0.006072532 -0.484488124
```

14.5.2 In beliebige Diagrammbereiche zeichnen

Die Voreinstellungen für ein device legen fest, dass zusätzliche Elemente nur in der Plot-Region gezeichnet werden können (vgl. Abschn. 14.1.1, Abb. 14.1). Diese *clipping* genannte Beschränkung kann über das Argument `xpd=NA` von `par()` aufgehoben werden (Abb. 14.9). Für zusätzliche Grafikelemente müssen auch außerhalb der Plot-Region letztlich User-Koordinaten verwendet werden, die durch die bereits dargestellten Daten definiert sind. Die einzelnen Device-Regionen besitzen jedoch auch eigene Koordinatensysteme, die es erlauben, allgemeingültige Koordinaten zur Plazierung von Elementen zu verwenden, die unabhängig von den konkreten Daten sind: Die Ränder jeder Device-Region definieren dafür jeweils ein Rechteck mit (x, y) -Koordinaten im Bereich 0–1, wobei $(0, 0)$ die linke untere und $(1, 1)$ die rechte obere Ecke bezeichnet. Diese Koordinaten müssen vor der Darstellung eines Elements in User-Koordinaten umgewandelt werden, wozu `cnvrt.coords()` aus dem Paket `Hmisc` dient.

```
> cnvrt.coords(x=(x-Koordinaten), y=(y-Koordinaten),
+                 input=c("usr", "plt", "fig", "dev", "tdev"))
```

Unter `x` und `y` sind die x bzw. y -Koordinaten der einzuziehnenden Elemente jeweils als Vektor einzutragen. Auf welches Koordinatensystem sie sich beziehen, legt das Argument `input` fest. Dabei steht "`usr`" für User-Koordinaten, "`plt`" für Koordinaten der Plot-Region, entsprechend "`fig`" für die Figure-Region, "`dev`" für die Device-Region ohne die äußeren Ränder und "`tdev`" für die Device-Region inkl. dieser Ränder.

`cnvrt.coords()` gibt eine Liste zurück, die für jedes Koordinatensystem eine Komponente mit den konvertierten Koordinaten besitzt. Die Komponenten tragen dabei dieselben Namen wie die möglichen Werte für das Argument `input`. Jede Komponente ist wiederum eine Liste mit den Komponenten `x` und `y` für die (x, y) -Koordinaten.

```
# clipping ausschalten und äußere Ränder setzen
> par(xpd=NA, oma=c(2, 2, 2, 2))
> plot(rnorm(10), xlab=NA, ylab=NA, pch=20)           # Streudiagramm
> library(Hmisc)                                     # für cnvrt.coords()

# Figure-Region links-unten
> pt1 <- cnvrt.coords(0, 0, input="fig")             # Figure-Koordinaten
> pt1$usr                                           # User-Koordinaten
$x
[1] -0.9669358

$y
[1] -3.389251

# Kreuz und zugehörigen Text einfügen
> points(pt1$usr, pch=4, lwd=5, cex=5, col="darkgray")
> text(pt1$usr$x + 0.5, pt1$usr$y, adj=c(0, 0), cex=1.5,
+       labels="Kreuz links-unten Figure-Region")

# gesamte Device-Region links-oben und rechts-unten
```

```
> pt2 <- cnvrt.coords(c(0.05, 0.95), c(0.95, 0.05), input="tdev")
> pt2$usr                                # User-Koordinaten
$x
[1] -1.064920 11.281049

$y
[1] 2.163520 -3.433674

# Pfeil und Text einfügen
> arrows(x0=pt2$usr$x[1], y0=pt2$usr$y[1], x1=pt2$usr$x[2],
+         y1=pt2$usr$y[2], lwd=4, code=3, angle=90, lend=2, col="darkgray")

> text(pt2$usr$x[1] + 0.5, pt2$usr$y[1], adj=c(0, 0), cex=1.5,
+       labels="Pfeil über gesamte Device-Region")
```

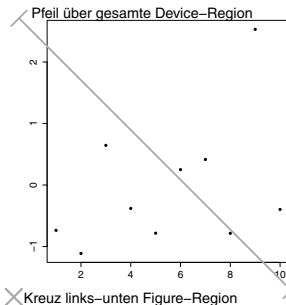


Abbildung 14.9: Grafikelemente an beliebiger Stelle eines Diagramms einfügen

14.5.3 Punkte

```
> points(x=<x-Koordinaten>, y=<y-Koordinaten>, type=<Option>)
> matpoints(x=<Matrix>, y=<Matrix>, type=<Option>)
```

Ähnlich wie `plot()` fügt `points()` einem geöffneten Diagramm Punkte hinzu, die Argumente `x`, `y` und `type` stimmen in ihrer Bedeutung mit jenen von `plot()` überein. Analog zu `matplot()` können mit `matpoints()` für `x`- und `y`-Koordinaten separate Matrizen angegeben und so gleichzeitig mehrere Datenreihen spezifiziert werden.

```
> xA <- seq(-15, 15, length.out=200)
> yA <- sin(xA) / xA                                # Sinc-Funktion
> plot(xA, yA, type="l", xlab="x", ylab="sinc(x)",
+       main="Punkte und Linien einfügen", lwd=1.6)

> xB <- seq(-15, 15, length.out=30)
> yB <- sin(xB) / xB
> points(xB, yB, col="red", pch=20)
```

14.5.4 Linien

```
> lines(x=<x-Koordinaten>, y=<y-Koordinaten>, type="Option")
> matlines(x=<Matrix>, y=<Matrix>, type="Option")
```

Datenpunkte verbindende Linien werden mit `lines()` analog zu Punkten erstellt. Dabei geben die Vektoren `x` und `y` die Koordinaten der zu verbindenden Punkte an, das Argument `type` bestimmt wie in `plot()` die genaue Art der Linien.¹⁰ `matlines()` arbeitet wie `matpoints()`, die (x, y) -Koordinaten können also für mehrere Datenreihen gleichzeitig in Form jeweils einer Matrix an die Argumente `x` und `y` übergeben werden (Abb. 14.10).

```
> yC <- sin(pi * xA) / (pi * xA) # normierte Sinc-Funktion
> lines(xA, yC, col="blue", type="l", lwd=1.6)

> abline(a=<y-Achsenabschnitt>, b=<Steigung>,
+         h=<y-Koordinate>, v=<x-Koordinate>, coef=<Vektor>)
```

Mit `abline()` werden Geradenabschnitte in ein Diagramm gezeichnet, die auf unterschiedliche Art spezifizierbar sind. Geraden können über die Gleichung $Y = bX + a$ mit den Parametern `a` für den Schnittpunkt mit der y -Achse und `b` für die Steigung beschrieben werden. Diese beiden Parameter erwartet alternativ auch das Argument `coef` in Form eines Vektors mit zwei Elementen. Statt dieser Parameter akzeptiert `abline()` auch ein mit `lm()` erstelltes Objekt, um die angepasste Vorhersagegerade einer einfachen linearen Regression darzustellen (vgl. Abschn. 6.2). Ein über die gesamte Breite der Plot-Region gehender horizontaler Geradenabschnitt kann über das Argument `h` in seiner y -Koordinate bezeichnet werden, ein vertikaler entsprechend über das Argument `v` in seiner x -Koordinate. Es lassen sich mehrere horizontale oder vertikale Geradenabschnitte mit nur einem Aufruf von `abline()` erzeugen, etwa um ein Gitter zu zeichnen. Dafür werden sowohl `h` als auch `v` gleichzeitig verwendet und für sie Vektoren von Koordinaten angegeben (Abb. 14.10).

```
> abline(h=0, v=0, col="green", lwd=1.6)

> grid(nx=NULL, ny=nx)
```

Die Argumente `nx` und `ny` von `grid()` bestimmen, wie viele Elemente in das einzufügende Gitter senkrecht zur x - und y -Achse eingezeichnet werden. Auf `NULL` gesetzt bildet das Gitter die Fortsetzung der Wertemarkierungen der zugehörigen Achse. Sollen keine Gitterelemente senkrecht zu einer Achse gezeichnet werden, ist das entsprechende Argument auf `NA` zu setzen. Für quadratische Gitterfelder muss beim Erstellen des Diagramms z. B. mit `plot()` das Seitenverhältnis mit dem Argument `asp=1` kontrolliert werden.

Im folgenden Diagramm wird das Ergebnis einer Regression grafisch veranschaulicht, insbesondere die Residuen als Differenz der tatsächlichen zu den vorhergesagten Werten (Abb. 14.11).

¹⁰Das Aussehen von Linienenden sowie das Verhalten von sich treffenden Endpunkten bestimmen die Argumente `lend` und `ljoin` von `par()`.

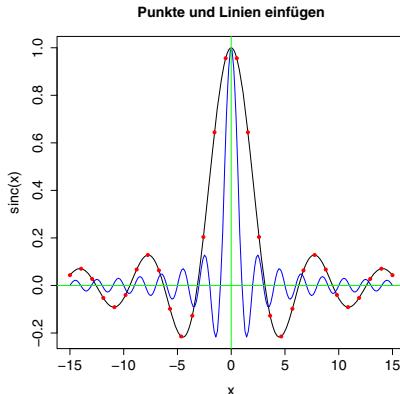


Abbildung 14.10: Einfügen von Diagrammelementen: Punkte und Linien

```

> height <- rnorm(20, 175, 7)                                # Prädiktor
> weight <- 0.5*height + 10 + rnorm(20, 0, 4)                # Kriterium
> fit      <- lm(weight ~ height)                            # Regression
> pred     <- fitted(fit)                                  # Vorhersage
> plot(weight ~ height, asp=1, col="blue", pch=16,
+       main="Gitter, Segmente und Pfeile einfügen")          # Regressionsgerade

> abline(fit, lwd=2)
> grid(lwd=2, col="darkgray")

```

Einzelne Liniensegmente können mit `segments()` eingezeichnet werden. Dazu werden in der Funktion die (x, y) -Koordinaten der Endpunkte der Segmente angegeben.

```
> segments(x0=<x Start>, y0=<y Start>, x1=<x Ziel>, y1=<y Ziel>)
```

Unter `x0` und `y0` sind die Koordinaten des Startpunkts zu nennen, von dem ausgehend das Segment gezeichnet wird. Unter `x1` und `y1` wird der Zielpunkt des Segments mit seinen Koordinaten angegeben. Wenn mehrere Linien zu zeichnen sind, lassen sich die Koordinaten auch als Vektoren eingeben.

```
# Differenz Vorhersage-Kriterium (Residuen)
> segments(x0=height, y0=pred, x1=height, y1=weight, col="gray")
```

Ähnlich wie Liniensegmente können auch Pfeile mit `arrows()` in ein Diagramm eingezeichnet werden.

```
> arrows(x0=<x Start>, y0=<y Start>, x1=<x Ziel>, y1=<y Ziel>,
+         length=0.25, angle=30, code=2)
```

Unter `x0` und `y0` werden die Koordinaten des Startpunkts eingegeben, von dem ausgehend der Pfeil gezeichnet wird. Unter `x1` und `y1` wird der Zielpunkt des Pfeils mit seinen Koordinaten

angegeben. Wenn mehrere Pfeile eingefügt werden sollen, können die Koordinaten auch als Vektoren eingegeben werden. Das Argument `length` kontrolliert die Länge der Pfeilspitzen in der Einheit inch, ihr Winkel wird über `angle` in Grad festgelegt. In der Voreinstellung `code=2` wird eine Pfeilspitze am Zielpunkt eingezeichnet, mit `code=1` am Startpunkt, mit `code=3` an beiden Enden (Abb. 14.11).

```
> arrows(x0=c(height[1]-3,    height[3]), y0=c(weight[1], weight[3]+3),
+         x1=c(height[1]-0.5, height[3]), y1=c(weight[1], weight[3]+0.5),
+         col="red", lwd=2)

> arrows(x0=height[4]+0.1*(height[8]-height[4]),
+         y0=weight[4]+0.1*(weight[8]-weight[4]),
+         x1=height[4]+0.9*(height[8]-height[4]),
+         y1=weight[4]+0.9*(weight[8]-weight[4]),
+         code=3, col="red", lwd=2)
```

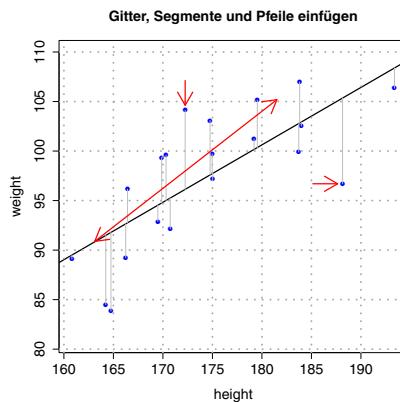


Abbildung 14.11: Einfügen von Diagrammelementen: Gitter, Liniensegmente und Pfeile

14.5.5 Polygone

Einem Diagramm kann mit `box(which='Region')` an verschiedenen Stellen ein rechteckiger Rahmen hinzugefügt werden. Das Argument `which` bestimmt, ob der Rahmen um die Plot-Region ("plot") oder das gesamte Diagramm ("figure") gezeichnet werden soll. Andere mögliche Werte sind "inner" und "outer", die in der Voreinstellung identische Ergebnisse zu "figure" erzielen und sich nur bei geänderten Seitenrändern anders verhalten.

`rect()` erzeugt beliebig dimensionierte Rechtecke, die sich in der Plot-Region frei plazieren lassen.

```
> rect(xleft=<x-Koord. links>, ybottom=<y-Koord. unten>,
+       xright=<x-Koord. rechts>, ytop=<y-Koord. oben>, border=NULL)
```

Die Argumente `xleft`, `ybottom`, `xright` und `ytop` akzeptieren Vektoren, die jeweils die Koordinaten der linken (x), unteren (y), rechten (x) und oberen Seiten (y) der zu zeichnenden Rechtecke enthalten. Soll kein Rahmen um ein Rechteck gezogen werden, ist `border=NA` zu setzen (Abb. 14.12; Hoffman, 2000).

```

> n      <- 7                                # Anzahl an Zeilen und Spalten
> len    <- 1/n                             # Kantenlänge eines Quadrats

# Farbverlauf im RGB Farbsystem erstellen (Blau-Anteil ist überall 0)
> colsR <- rep(seq(0.9, 0.2, length.out=n), each=n)      # Rot-Anteil
> colsG <- rep(seq(0.9, 0.2, length.out=n), times=n)     # Grün-Anteil
> cols   <- rgb(colsR, colsG, 0)                  # Farben in RGB

# Koordinaten der Quadrate festlegen
> xLeft  <- rep(seq(0, 1-len, by=len), times=n)        # x-Koord. links
> yBot   <- rep(seq(0, 1-len, by=len), each=n)          # y-Koord. unten
> xRight <- rep(seq(len, 1,       by=len), times=n)      # x-Koord. rechts
> yTop   <- rep(seq(len, 1,       by=len), each=n)        # y-Koord. oben

# zunächst ein leeres Diagramm erzeugen, dann Rechtecke einzeichnen
> par(oma=c(1, 1, 1, 1), mar=c(0, 0, 1, 0))           # Ränder ändern
> plot(c(0, 1), c(0, 1), axes=FALSE, type="n", asp=1, main="Farbverlauf")
> rect(xLeft, yBot, xRight, yTop, border=NA, col=cols)

# zwei Quadrate mit ihren hexadezimalen Farbwerten beschriften
> idx    <- c(10, 27)
> xText <- xLeft[idx] + (xRight[idx] - xLeft[idx])/2
> yText <- yBot[idx]  + (yTop[idx]  - yBot[idx])/2
> text(xText, yText, labels=cols[idx])

# für zweites Diagramm Farben in zufällige Reihenfolge bringen
> shuffled <- sample(along=cols, length(cols), replace=FALSE)
> idxS     <- c(which(shuffled == idx[1]), which(shuffled == idx[2]))
> plot(c(0, 1), c(0, 1), axes=FALSE, type="n", asp=1,
+       main="Dieselben Farben zufällig angeordnet")

> rect(xLeft, yBot, xRight, yTop, border=NA, col=cols[shuffled])

# die zwei Quadrate mit ihren hexadezimalen Farbwerten beschriften
> xTextS <- xLeft[idxS] + (xRight[idxS] - xLeft[idxS])/2
> yTextS <- yBot[idxS]  + (yTop[idxS]  - yBot[idxS])/2
> text(xTextS, yTextS, labels=cols[idx])

```

`polygon()` erzeugt Vielecke beliebiger Gestalt, deren Eckpunkte über ihre jeweiligen (x, y)-Koordinaten zu definieren sind und in Form von Vektoren an `x` und `y` übergeben werden können. Das Polygon wird geschlossen, indem R den ersten und letzten Punkt miteinander verbindet.¹¹

¹¹Mit `polypath()` lassen sich auch Polygone mit sich selbst überschneidenden Kanten zeichnen.

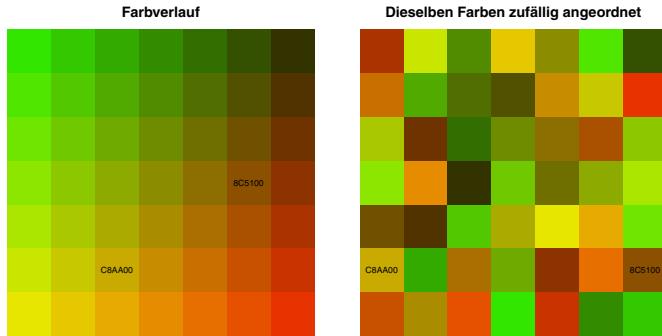


Abbildung 14.12: Mit `rect()` erzeugte Rechtecke zur Veranschaulichung eines Farbphänomens: Beide Grafiken sind aus denselben Quadraten zusammengesetzt, die jedoch – auch abgesehen vom Chevreul-Effekt links – in den beiden Konfigurationen unterschiedlich aussehen. Zwei jeweils links und rechts identische Farben sind mit ihrem Hexadezimalwert bezeichnet

Soll kein Rahmen gezogen werden, ist `border=NA` zu setzen. Mittels eng gesetzter Eckpunkte lassen sich auch runde Formen durch Polygone approximieren.

```
> polygon(x=(x-Koordinaten), y=(y-Koordinaten), border=NULL)
```

Das folgende Beispiel illustriert die Beziehung zwischen der Fläche unter der Kurve der Dichtefunktion einer Normalverteilung und der Wahrscheinlichkeit von Intervallen, wie sie sich an der zugehörigen Verteilungsfunktion als Differenz der Funktionswerte der Intervallgrenzen ablesen lässt. Im Diagramm werden beide Funktionen gleichzeitig unter Verwendung unterschiedlicher y-Achsen dargestellt (Abb. 14.13).

```
> mu      <- 0                      # Erwartungswert
> sigma   <- 3                      # Streuung
> xLims  <- c(mu-4*sigma, mu+4*sigma)    # Grenzen x-Achse
> X       <- seq(xLims[1], xLims[2], length.out=100)      # x-Werte
> Y       <- dnorm(X, mu, sigma)          # Werte der Dichtefunktion
> selX   <- seq(mu-sigma, mu+sigma, length.out=100)        # [mu +- sigma]
> selY   <- dnorm(selX, mu, sigma)        # Werte im Intervall [mu +- sigma]
> cdf    <- pnorm(X, mu, sigma)          # Werte der Verteilungsfunktion

# Grafik leer mit breitem rechten Rand öffnen
> par(mar=c(5, 4, 4, 5))
> plot(X, Y, type="n", xlim=xLims-c(-2, 2), xlab=NA, ylab=NA,
+       main="Dichtefunktion und Verteilungsfunktion N(0, 3)")

# Rahmen, Fläche über Intervall [mu +- sigma] zeichnen
> box(which="plot", col="gray", lwd=2)
> polygon(c(selX, rev(selX)), c(selY, rep(-1, length(selX))),
+           border=NA, col="lightgray")
```

```

> lines(X, Y, lwd=2)                                # Dichtefunktion

# Verteilungsfunktion einzeichnen
> par(new=TRUE)                                     # folgendes Diagramm in bestehendes zeichnen
> plot(X, cdf, xlim=xLims-c(-2, 2), type="l", lwd=2, col="blue",
+       xlab="x", ylab=NA, axes=FALSE)

# Achsen und Hilfslinien einzeichnen
> axis(side=4, at=seq(0, 1, by=0.1), col="blue")
> segments(x0=c(mu-sigma, mu, mu+sigma), y0=c(-1, -1, -1),
+           x1=c(mu-sigma, mu, mu+sigma), y1=c(pnorm(mu-sigma, mu, sigma),
+           pnorm(mu, mu, sigma), pnorm(mu+sigma, mu, sigma)),
+           lwd=2, col=c("darkgreen", "red", "darkgreen"), lty=2)

> segments(x0=c(mu-sigma, mu, mu+sigma), y0=c(pnorm(mu-sigma, mu, sigma),
+           pnorm(mu, mu, sigma), pnorm(mu+sigma, mu, sigma)),
+           x1=xLims[2]+10, y1=c(pnorm(mu-sigma, mu, sigma),
+           pnorm(mu, mu, sigma), pnorm(mu+sigma, mu, sigma)),
+           lwd=2, col=c("darkgreen", "red", "darkgreen"), lty=2)

> arrows(x0=c(mu-sigma+0.2, mu+sigma-0.2), y0=-0.02, x1=c(mu-0.2, mu+0.2),
+         y1=-0.02, code=3, angle=90, length=0.05, lwd=2, col="darkgreen")

# zusätzliche Beschriftungen
> mtext(text="F(x)", side=4, line=3)
> rect(-8.5, 0.92, -5.5, 1.0, col="lightgray", border=NA)
> text(-7.2, 0.9, labels="Wahrscheinlichkeit")
> text(-7.1, 0.86, expression(des~~Intervalls~~group("[",
+     list(-sigma, sigma), "]")))
> text(mu-sigma/2, 0,      expression(sigma), col="darkgreen", cex=1.2)
> text(mu+sigma/2, 0,      expression(sigma), col="darkgreen", cex=1.2)
> text(mu+0.5,      0.02, expression(mu),    col="red",       cex=1.2)

```

14.5.6 Funktionsgraphen

```
> curve(expr=<Funktion>, from=<Zahl>, to=<Zahl>, n=101, add=FALSE)
```

Die High-Level-Funktion `curve()` dient dazu, Graphen von beliebigen Funktionen mit einer Veränderlichen zu erstellen. Dabei kann die darzustellende Funktion als Funktionsgleichung mit `x` als Variable spezifiziert werden (z. B. `dnorm(x, mean=1, sd=1)` oder `x^2 + 10`). Hier wird also dieselbe Notation benutzt, wie um aus einem bestehenden Vektor `x` einen neuen Vektor mit den Funktionswerten zu generieren. Als Kurzform lässt sich auch nur der Name einer Funktion angeben (z. B. `dnorm`), die dann mit den Voreinstellungen für ihre Argumente aufgerufen wird. In welchem Wertebereich die Funktion ausgewertet werden soll, bestimmen

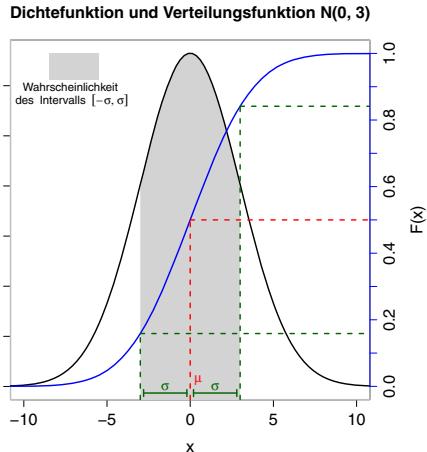


Abbildung 14.13: Diagrammelemente einfügen: Rechtecke, Polygone, Achsen, Text und Symbole

die Argumente `from` und `to`. Dabei legt das Argument `n` die Anzahl der Stützstellen fest, an wie vielen gleichabständigen Stellen in diesem Bereich also Funktionswerte bestimmt und eingezeichnet werden. Um den Funktionsgraphen dem aktuell aktiven device hinzuzufügen, ohne dessen Inhalten zu löschen, ist `add=TRUE` zu setzen (Abb. 14.14). Als auf der Konsole unsichtbaren Rückgabewert liefert `curve()` eine Liste mit den (x, y) -Koordinaten der gezeichneten Punkte.

```
> mu      <- 0                      # Normalverteilung Erwartungswert
> sigma   <- 2                      # theoretische Streuung
> curve(dnorm(x, mean=1, sd=1), from=-7, to=7, col="blue", lwd=2)
> curve((1/(sigma*sqrt(2*pi))) * exp(-0.5*((x-mu)/sigma)^2)),
+       add=TRUE, lwd=2, lty=2)
```

14.5.7 Text und mathematische Formeln

```
> title(main="<Titel>", sub="<Untertitel>)
```

Der Diagrammtitel lässt sich in High-Level-Grafikfunktionen über das Argument `main="<Name>"`, ein Untertitel über `sub="<Name>"` hinzufügen. Soll der Diagrammtitel nachträglich festgelegt werden, kann die separat aufzurufende Low-Level-Funktion `title()` Verwendung finden, die ihrerseits `main` und `sub` als Argumente besitzt. Die Escape-Sequenz `\n` dient innerhalb einer Zeichenkette als Symbol für den Zeilenwechsel (Abb. 14.14).

```
> title(main="zwei Normalverteilungskurven", sub="Untertitel")
```

Eine Legende zur Erläuterung der verwendeten Symbole erstellt `legend()`.

```
> legend(x=<x-Koordinate>, y=<y-Koordinate>, legend="<Text>",
+         col=<Farben>, lty=<Linientypen>, lwd=<Linienstärken>,
+         pch=<Symbole>)
```

Die Legende wird entweder über die Angabe von (x, y) -Koordinaten für die Argumente `x` und `y` positioniert, oder durch Nennung eines der Schlüsselwörter "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" oder "center" für `x`. Der Legendentext selbst ist als Vektor von Zeichenketten an `legend` zu übergeben, wobei jedes Element dieses Vektors einen Legendeneintrag definiert. Welches Aussehen dem Symbol neben einem Eintrag verliehen wird, kontrollieren die Argumente `col` (Farbe), `lty` (Linientyp), `lwd` (Linienstärke) und `pch` (Symbol), vgl. Abschn. 14.3.1. Für diese Argumente ist jeweils ein Vektor derselben Länge wie `legend` einzugeben, wobei `NA` Einträge bedeuten, dass die definierte Eigenschaft nicht auf das zugehörige Legendsymbol zutrifft.

Sind in einem Diagramm etwa sowohl zwei Punkt-Reihen als auch zwei Linien enthalten, würde die Kombination von `pch=c(19, 20, NA, NA)` und `lty=c(NA, NA, 1, 2)` bewirken, dass die ersten beiden Legendeneinträge mit den Symbolen 19 und 20 dargestellt werden, die letzten beiden Einträge mit den Linientypen 1 und 2.

```
> legend(x="topleft", legend=c("N(1, 1)", "N(0, 2)",
+     col=c("blue", "black"), lty=c(1, 2))
```

Mit `text()` lässt sich allgemein Text an beliebiger Stelle in die Plot-Region eines Diagramms einfügen.

```
> text(x=<x-Koordinaten>, y=<y-Koordinaten>, adj=<Positionskorrektur>,
+       labels=<Name>, srt=<Rotationswinkel>)
```

Zunächst sind mit den Argumenten `x` und `y` die Koordinaten des Texts festzulegen. Sollen mehrere Textelemente gleichzeitig eingefügt werden, sind hier Vektoren zu übergeben. In der Voreinstellung beziehen sich die Koordinaten auf den Mittelpunkt des Texts, was jedoch über `adj` veränderbar ist: Durch einen Vektor mit zwei Elementen im Intervall $[0, 1]$ kann der Bezugspunkt der (x, y) -Koordinaten vom linken unteren Textrand (`c(0, 0)`) zum rechten oberen Textrand (`c(1, 1)`) verschoben werden, Voreinstellung ist `c(0.5, 0.5)` für die Textmitte. Die Texte selbst müssen dem Argument `labels` in Form eines Vektors von Zeichenketten übergeben werden. Das Argument `srt` erwartet eine Winkelangabe in Grad und erlaubt es, den Text um den mit `adj` definierten Drehpunkt zu rotieren.

```
> text(x=3.6, y=0.35, labels="Normalverteilung\nN(1, 1)")
> text(x=-3.5, y=0.1, labels="N(0, 2)")
```

Sollen Textelemente nicht innerhalb der Plot-Region, sondern an deren äußere Ränder geschrieben werden, kann `mtext()` eingesetzt werden.

```
> mtext(text=<Name>, side=<Nummer>, line=<Nummer>)
```

Während das Argument `text` die darzustellenden Texte in Form eines Vektors von Zeichenketten akzeptiert, bestimmt `side` mit einem Vektor der Zahlen 1–4, an welcher Seite der Text erscheinen soll. Die 1 steht dabei für unten, 2 für links, 3 für oben (Voreinstellung) und 4 für rechts. Die Orientierung des Texts ist immer parallel zum Rand, an dem der Text steht. Das Argument `line` legt in Form eines Vielfachen der Linienhöhe fest, wie weit außen der Text dargestellt wird, wobei 0 dem Rand der Plot-Region entspricht.

```
> mtext(text="Wahrscheinlichkeitsdichte", side=3)
```

In allen Funktionen zum Einfügen von Text können mit Hilfe einer an das Textsatzsystem L^AT_EX angelehnten Syntax auch jedwede Art von Symbolen (griechische Buchstaben, mathematische Sonderzeichen, etc.) und mathematische Formeln definiert werden (Ligges, 2002). Zu diesem Zweck wird eine Formel in Textform an `expression(<Ausdruck>)` übergeben und das Ergebnis in den Funktionen zum Einfügen von Text eingesetzt. `<Ausdruck>` enthält dabei etwa Text (ohne Anführungszeichen), lateinische Umschreibungen griechischer Buchstaben (`theta`), Summenzeichen (`sum(x[i], i==1, n)`) Brüche (`frac(<Zähler>, <Nenner>)`) oder Wurzel-symbole (`sqrt(<Radikand>, <Wurzelexponent>)`). Zeilenumbrüche mit der `\n` Escape-Sequenz sind dagegen nicht möglich – stattdessen müssen mehrere Zeilen durch mehrere `text(x, y, ~\rightarrow expression())` Befehle mit entsprechend unterschiedlichen Koordinaten realisiert werden. Eine Einführung in die Verwendung enthält die Hilfe-Seite `?plotmath`, weitere Veranschaulichungen zeigt `demo(plotmath)`.

```
> text(-4, 0.3, expression(frac(1, sigma*sqrt(2*pi)) ~~
+                               exp * bgroup("(", -frac(1, 2) ~~
+                               bgroup("(", frac(x-mu, sigma), ")")^2, ")")))

```

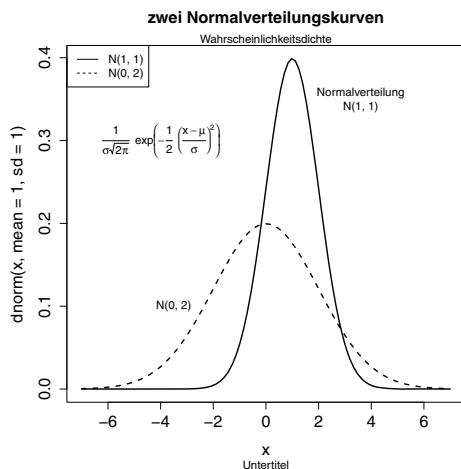


Abbildung 14.14: Diagrammelemente einfügen: Funktionsgraphen, Text, Legende und mathematische Formeln

14.5.8 Achsen

Ob durch die Darstellung von Datenpunkten in einem Diagramm automatisch auch Achsen generiert werden, kontrollieren in High-Level-Grafikfunktionen die Argumente `xaxt` für die *x*-Achse, `yaxt` für die *y*-Achse und `axes` für beide Achsen gleichzeitig. Während für `xaxt` und `yaxt` der Wert "n" übergeben werden muss, um die Ausgabe der entsprechenden Achse zu unterdrücken, akzeptiert `axes` dafür den Wert FALSE.

Für eine feinere Kontrolle über den Wertebereich sowie über Aussehen und Lage der Wertemarkierungen der Achsen empfiehlt es sich, ihre automatische Generierung zunächst mit `axes=FALSE` zu unterdrücken. Nachdem das Diagramm erstellt wurde, können dann mit dem Befehl `axis()` samt seiner Argumente zur Formatierung und Positionierung Achsen hinzugefügt werden (Abb. 14.13, 14.15).

```
> axis(side=<Nummer>, at=<Markierungen>, labels=<Wertebeschriftungen>",
+      pos=<Position>)
```

Achsen lassen sich an allen Diagrammseiten darstellen, was über das Argument `side` kontrolliert wird. Mögliche Werte sind 1 (unten, x -Achse), 2 (links, y -Achse), 3 (oben, alternative x -Achse) und 4 (rechts, alternative y -Achse). Die Wertemarkierungen der Achse lassen sich über `at` in Form eines Vektors festlegen.¹² Das Argument `labels` bestimmt die Beschriftung dieser Markierungen und erwartet einen numerischen Vektor oder einen Vektor von Zeichenketten. Die Orientierung dieser Markierungen legt das Argument `las` von `par()` fest. Soll die Position der Achse nicht an den Rändern der Plot-Region liegen, kann sie auch in Form einer Koordinate über das Argument `pos` definiert werden. Ist die Achse eine horizontale (`side=1` oder 3), wird der Wert für `pos` als y -Koordinate der Achse interpretiert, andernfalls (`side=2` oder 4) als deren x -Koordinate.

```
> vec <- seq(from=-2*pi, to=2*pi, length.out=200)
> mat <- cbind(sin(vec), cos(vec), tan(vec))
> mat <- ifelse(abs(mat) > 2, NA, mat)           # Werte > 2 auf NA setzen
> matplot(vec, mat, lwd=2, col=c(12, 14, 17), type="l", , lty=c(1, 2, 4),
+           xaxt="n", xlab=NA, ylab=NA, main="Trigonometrische Funktionen")

> xTicks <- seq(from=-2*pi, to=2*pi, by=pi/2)
> xLabels <- c("-2*pi", "-3*pi/2", "-pi", "-pi/2", "0", "pi/2", "pi",
+              "3*pi/2", "2*pi")

> axis(side=1, at=xTicks, labels=xLabels)
> abline(h=c(-1, 0, 1), v=seq(from=-3*pi/2, to=3*pi/2, by=pi/2),
+          col="gray", lty=3, lwd=2)

> abline(h=0, v=0, lwd=2)
> legend(x="bottomleft", legend=c("sin(x)", "cos(x)", "tan(x)"),
+          lty=c(1, 2, 4), col=c(12, 14, 17))
```

14.5.9 Fehlerbalken

Fehlerbalken werden zusätzlich zu Kennwerten von Variablen vor allem in Säulen- und Punktdiagrammen eingezeichnet, um die Variabilität der Daten auszudrücken. Als Maß der Variabilität kann dabei u. a. die Breite eines statistischen Konfidenzintervalls für einen Parameter (z. B. den Erwartungswert) oder ein deskriptives Maß wie die Streuung verwendet werden.¹³

¹²Unterbrochene Achsen können mit `axis.break()` aus dem `plotrix` Paket eingezeichnet werden.

¹³Für Konfidenzellipsen als Maß für die Variabilität zweidimensionaler Daten vgl. `confidenceEllipse()` aus dem `car` Paket und Abschn. 14.6.8.

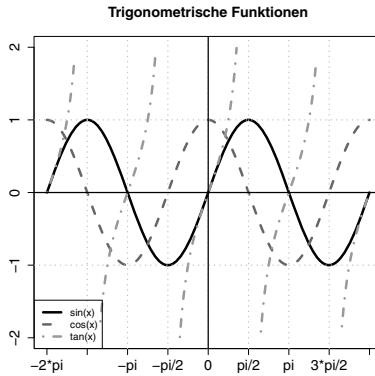


Abbildung 14.15: Diagrammachsen mit `axis()` anpassen

Das Paket `DescTools` enthält die Funktion `ErrBars()`, die Fehlerbalken einem bestehenden Diagramm hinzufügt. Das vertikale Zentrum der Fehlerbalken wird als Punkt gezeichnet, so dass es nicht unbedingt notwendig ist, zusätzlich Säulen oder Punkte zur Veranschaulichung des Parameters zu zeichnen, dessen Variabilität über einen Fehlerbalken dargestellt wird (Abb. 14.16).

```
> ErrBars(from=<y-Koord. unten>, to=<y-Koord. oben>, pos=<x-Koord.>,
+           horiz=FALSE)
```

Unter `from` und `to` werden die y -Koordinaten der unteren bzw. oberen Grenzen der Fehlerbalken definiert. Für `pos` sind die x -Koordinaten der Fehlerbalken anzugeben. Mit `horiz` wird definiert, ob die Balken vertikal (Voreinstellung FALSE) oder horizontal (TRUE) zu zeichnen sind. Die Funktion verfügt über weitere Optionen zur Formatierung der Fehlerbalken bzgl. ihrer Farbe, Linienstärke, etc.

```
# Simulation von Messwerten in 4 Gruppen ohne Gruppeneffekt
> Nj <- c(15, 20, 18, 22)                                # Gruppengrößen
> P <- length(Nj)                                         # Anzahl Gruppen
> DV <- rnorm(sum(Nj), rep(c(30, 20, 25, 15), Nj), 6)    # Daten
> IV <- factor(rep(1:P, Nj))                             # Gruppenzugehörigkeit
> Mj <- tapply(DV, IV, mean)                            # Gruppenmittel
> Sj <- tapply(DV, IV, sd)                               # Gruppenstreuungen

# halbe Breite 95% t-Konfidenzintervalle für Erwartungswerte
> ciWidths <- qt(0.975, Nj-1) * Sj/sqrt(Nj)

# Punktdiagramm getrennt nach Gruppen
> stripchart(DV ~ IV, method="jitter", xaxt="n", xlab="Gruppe",
+             ylim=c(0, 40), main="Rohdaten und Konfidenzintervalle",
+             col="darkgray", pch=16, vert=TRUE)
```

Kapitel 14 Diagramme erstellen

```

> library(DescTools)                                # für ErrBars()
> ErrBars(from=Mj-ciWidths, to=Mj+ciWidths, pos=1:P, length=0.1,
+         col="blue", col.pch="blue", lwd=2, pch=19)

> axis(side=1, at=1:P, labels=LETTERS[1:P])          # Gruppenbezeichnungen

Es können auch simultan Fehlerbalken für mehrere Gruppen dargestellt werden, die ähnlich einem gruppierten Säulendiagramm aufgebaut sind. Dabei wird die Gruppierung durch die Wahl der  $x$ -Koordinaten kontrolliert. Hier soll die Streuung die Länge der Fehlerbalken bestimmen.

> Mj1 <- c(2, 3, 6, 3, 5)                         # Mittelwerte Gruppe 1
> Sj1 <- c(1.7, 1.8, 1.7, 1.9, 1.8)                # Streuungen Gruppe 1
> Mj2 <- c(4, 3, 2, 1, 3)                          # Mittelwerte Gruppe 2
> Sj2 <- c(1.4, 1.7, 1.7, 1.3, 1.5)                # Streuungen Gruppe 2
> Q <- length(Mj1)                                 # Anzahl Mittelwerte
> xOff <- 0.1                                     # horizontaler Offset zwischen Werten einer Gruppe

# Mittelwerte
> plot(c((1:Q)-xOff, (1:Q)+xOff), c(Mj1, Mj2), pch=19,
+       main="Mittelwerte und SDs im 5x2 Design",
+       xlab="Faktor A", ylab="Mittelwert",
+       col=rep(c("blue", "red"), each=5), ylim=c(0, 8))

# Fehlerbalken
> ErrBars(from=c(Mj1, Mj2) - c(Sj1, Sj2), to=c(Mj1, Mj2) + c(Sj1, Sj2),
+           pos=c((1:Q)-xOff, (1:Q)+xOff), lty=rep(1:2, each=Q),
+           col=rep(c("blue", "red"), each=Q),
+           col.pch=rep(c("blue", "red"), each=Q), pch=19)

> legend(x="topleft", legend=c("B-1", "B-2"), pch=c(19, 19),
+          col=c("blue", "red"))

```

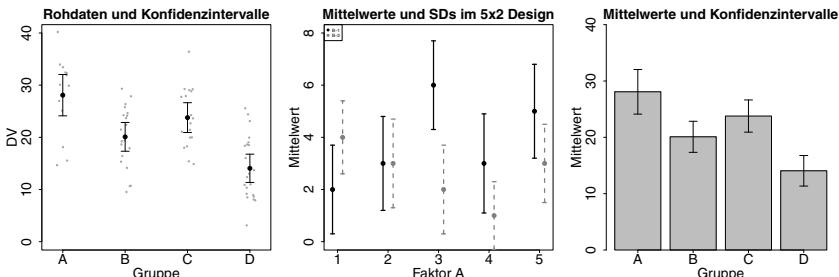


Abbildung 14.16: Fehlerbalken mit `ErrBars()` oder als Pfeile einfügen

Um selbst konstruierte Fehlerbalken in ein Diagramm einzufügen, können mit `arrows()` erstellte Pfeile „zweckentfremdet“ werden (vgl. Abschn. 14.5.4). Pfeilen lässt sich das Aussehen von Fehlerbalken geben, indem an beiden Enden Pfeilspitzen gezeichnet werden (`code=3`), für deren

Winkel 90° zu wählen (`angle=90`) und deren Länge zu verkürzen ist (`length=0.1`, Abb. 14.16). Die benötigten x -Koordinaten der Fehlerbalken sind im von `barplot()` zurückgegebenen Vektor enthalten. Bei einem gruppierten Säulendiagramm handelt es sich stattdessen um eine Matrix. Die Höhe der Säulen liefert den vertikalen Mittelpunkt der Fehlerbalken und ergibt sich direkt aus den Daten, ebenso die y -Koordinaten der Endpunkte der Fehlerbalken als Grenzen des Konfidenzintervalls für den geschätzten statistischen Kennwert.

```
# Säulendiagramm
> barsX <- barplot(height=Mj, xaxt="n", xlab="Gruppe", ylim=c(0, 40),
+                      ylab="Mittelwert", main="Mittelwerte und Konfidenzintervalle")

> axis(side=1, at=barsX, labels=LETTERS[1:P])          # Gruppenbezeichnungen
> ciLo <- Mj - ciWidths                                # Fehlerbalken untere \ 
→Grenze
> ciHi <- Mj + ciWidths                                # Fehlerbalken obere \
→Grenze

# Fehlerbalken mit arrows()
> arrows(x0=barsX, y0=ciLo, x1=barsX, y1=ciHi,
+         code=3, angle=90, length=0.1, col="blue")
```

14.5.10 Rastergrafiken

Rastergrafiken definieren ein Bild als Ansammlung diskreter Bildpunkte (*pixel – picture elements*), während Vektorgrafiken dies durch eine strukturelle Beschreibung der im Bild dargestellten Objekte tun. Für jeden Bildpunkt speichern Rastergrafiken den Farbwert, den das Bild an diesem Punkt besitzen soll. Rastergrafiken werden auch als Bitmap- bzw. Pixel-Grafiken bezeichnet und lassen sich in R in zwei Schritten erzeugen und darstellen: Zunächst ist eine Matrix mit Farbwerten zu füllen, wobei jedes ihrer Elemente einen Bildpunkt festlegt. `rasterImage()` fügt das so definierte Bild dann einem bestehenden Diagramm hinzu (Abb. 14.17).¹⁴

```
> rasterImage(image=<Farb-Matrix>, xleft=<x-Koord.>, ybottom=<y-Koord.>,
+               xright=<x-Koord.>, ytop=<y-Koord.>, angle=<Drehwinkel>,
+               interpolate=TRUE)
```

An `image` ist eine Matrix mit Farbwerten zu übergeben (vgl. Abschn. 14.3.2). Anstatt die Farbwerte direkt in eine Matrix zu schreiben, lässt sich auch eine Matrix bzw. ein array mit Zahlen im Bereich von 0–1 mit Hilfe von `as.raster()` in eine solche Farbwert-Matrix umwandeln. Wird dabei eine Matrix übergeben, symbolisiert jede Zahl den Grauwert eines Bildpunkts. Für farbige Bilder ist ein array mit drei Ebenen zu verwenden: Die Matrix der ersten Ebene definiert den Rot-Anteil, die der zweiten Ebene den Grün- und die der dritten Ebene den Blau-Anteil der Farbe jedes Bildpunkts.

Mit den Argumenten `xleft` und `ybottom` werden die (x, y) -Koordinaten der Position im Diagramm definiert, an der die linke untere Bildecke liegen soll, mit `xright` und `ytop` entsprechend

¹⁴Als Grafikdatei vorhandene Bitmap-Bilder können mit Funktionen aus den Paketen `adimpro` (Polzehl & Tabelow, 2007) oder `EBImage` (Pau, Fuchs, Sklyar, Boutros & Huber, 2010) eingelesen werden.

die Koordinaten für die rechte obere Bildecke. `angle` erlaubt es mit einer Winkelangabe in Grad, die Rastergrafik gegen den Uhrzeigersinn um die linke untere Bildecke zu drehen. Der über die Koordinaten der Bildecken ausgewählte Bereich des Diagramms kann in der Darstellung letztlich mehr oder weniger Bildpunkte abdecken, als in `image` definiert sind. Das Bild muss deshalb auf den Abbildungsbereich gestreckt oder gestaucht werden. In der Voreinstellung `interpolate=TRUE` wird beim Strecken linear zwischen der Farbe vormals angrenzender Bildpunkte interpoliert, was in etwas weniger abrupten Farbabstufungen resultiert.

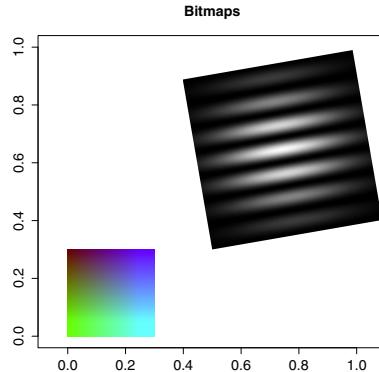


Abbildung 14.17: Rastergrafiken in einem Diagramm anzeigen

Im Beispiel soll ein Bild aus farbigen Rechtecken und zusätzlich eine Gabor-Funktion dargestellt werden, also eine orientierte zweidimensionale Cosinus-Funktion, deren Amplitude einer zweidimensionalen Normalverteilung folgt.

```
> pxSq <- 6                                # Anzahl Bildpunkte pro Achse
> colsR <- rep(0.4, pxSq^2)                  # Rot-Anteil: hier konstant
> colsG <- rep(seq(0, 1, length.out=pxSq), times=pxSq)      # Grün
> colsB <- rep(seq(0, 1, length.out=pxSq), each=pxSq)       # Blau

# array mit drei Ebenen: Rot, Grün, Blau
> arrSq <- array(c(colsR, colsG, colsB), c(pxSq, pxSq, 3))
> sqIm   - as.raster(arrSq)                  # Zahlen -> Farbwerte

# Gabor-Funktion
> pxGab <- 500                               # Anzahl Bildpunkte pro Achse
> alpha <- 0.5                                # Faktor für Winkel 2D-Cosinus
> beta  <- min(1-alpha, 1+alpha)              # für komplementären Winkel
> freq   <- 3.5                                # Frequenz 2D-Cosinus

# x- und y-Wertebereich für 2D-Cosinus und 2D-Normalverteilung
> vals <- rep(seq(-2*pi, 2*pi, length.out=pxGab), pxGab)

# Matrizen mit den x- bzw. y-Koordinaten aller Bildpunkte
```

```

> x    <- matrix(vals, nrow=pxGab, byrow=TRUE)
> y    <- matrix(vals, nrow=pxGab, byrow=FALSE)
> phi <- alpha*x + beta*y                      # konvexe Mischung der Koordinaten

# 2D-Cosinus skaliert auf Werte im Bereich [0, 1]
> cosMat <- 0.5*cos(freq*phi) + 0.5

# Werte unkorrelierter 2D-Normalverteilung
> library(mvtnorm)                            # für dmvnorm()
> mu      <- c(0, 0)                          # Zentroid
> sigma   <- diag(2)*10                        # Kovarianzmatrix
> gaussVal <- dmvnorm(cbind(c(x), c(y)), mu, sigma)

# als Matrix skaliert auf Werte im Bereich [0, 1]
> gaussMat <- matrix(gaussVal, nrow=pxGab) / max(gaussVal)

# Gabor = Produkt von 2D-Cos und 2D-NV, Zahlen -> Farbwerte
> gabIm <- as.raster(cosMat*gaussMat)

# öffne leeres Diagramm
> plot(c(0, 1), c(0, 1), type="n", main="Bitmaps", xlab="", ylab="", asp=1)

# füge Matrizen mit Farbwerten als Rastergrafiken ein
> rasterImage(sqIm, 0, 0, 0.3, 0.3, angle=0, interpolate=FALSE)
> rasterImage(gabIm, 0.5, 0.3, 1.1, 0.9, angle=10, interpolate=TRUE)

```

R verfügt auch über die grundlegenden Funktionen zur digitalen Signalverarbeitung, die sich insbesondere zur Bildanalyse und -manipulation eignen: Mit `fft()` ist die schnelle Fourier-Transformation und ihre Rücktransformation möglich, der Faltungsoperator ist in `convolve()` implementiert. Für weitere Funktionen vgl. die Pakete `adimpro` und `EBIImage`.

14.6 Verteilungsdiagramme

Verteilungsdiagramme dienen dazu, sich einen Überblick über die Lage und Verteilungsform der in einer Stichprobe erhobenen Daten zu verschaffen. Sie eignen sich damit auch zur Überprüfung der Daten auf Ausreißer oder unplausible Werte, die etwa aus Eingabefehlern herrühren können (vgl. Abschn. 4.2.7). Dies kann entweder anhand summarischer Kennwerte oder aber durch Darstellung von Einzelwerten, ggf. in vergrößerter Form, geschehen.

14.6.1 Histogramm und Schätzung der Dichtefunktion

Für Stichproben stetiger Variablen, die eine Vielzahl unterschiedlicher Werte enthalten, kann ein Histogramm als Sonderform eines Säulendiagramms für die Darstellung der empirischen Häufigkeitsverteilung verwendet werden. Histogramme stellen nicht die Häufigkeit einzelner

Werte, sondern die von Wertebereichen (disjunkten Intervallen) anhand von Säulen dar, zwischen denen kein Zwischenraum steht (Abb. 14.18).¹⁵

```
> hist(x=<Vektor>, breaks=<Grenzen>, freq=NULL)
```

Die Daten sind in Form eines Vektors `x` zu übergeben. Die Intervallgrenzen werden über das Argument `breaks` festgelegt, wobei mit einer einzelnen Zahl deren Anzahl und mit einem Vektor deren genaue Lage vorgegeben werden kann. In der Voreinstellung wird beides nach einem in der Hilfe beschriebenen Algorithmus entsprechend den Daten in `x` gewählt.¹⁶ Bei gleichabständigen Klassengrenzen werden in der Voreinstellung `freq=NULL` absolute Häufigkeiten angezeigt. Mit `freq=FALSE` entspricht der Flächeninhalt jeder Säule der relativen Häufigkeit des Intervalls, `TRUE` erzwingt absolute Häufigkeiten auch bei ungleichen Klassenbreiten. Der auf der Konsole nicht sichtbare Rückgabewert von `hist()` enthält in Form einer Liste u. a. Angaben zur Lage und Besetzung der verwendeten Intervalle.

```
> height <- rnorm(100, mean=175, sd=7)           # Körpergröße 100 Personen
> hist(height, xlab="height [cm]", ylab="N")
```

Für die individuelle Wahl der Klassengrenzen empfiehlt es sich, zunächst die Spannweite der Daten auszuwerten. Intervallgrenzen in regelmäßigen Abständen können dann z. B. mit `seq()` generiert werden. Werte, die genau auf einer Grenze liegen, werden immer der unteren Klasse zugeordnet, die Klassen sind also nach unten offene und nach oben geschlossene Intervalle.

```
# darzustellender Wertebereich
> fromTo <- round(range(height), -1) + c(-10, 10)
> limits <- seq(from=fromTo[1], to=fromTo[2], by=5)      # Intervallgrenzen
> hist(height, freq=FALSE, xlim=fromTo, xlab="height [cm]",
+       ylab="relative Häufigkeit", breaks=limits,
+       main="Histogramm und Normalverteilung")
```

Als weitere Information lässt sich im Anschluss an den Aufruf von `hist()` mit `rug()` auch die Lage der Einzelwerte mit darstellen. Dies geschieht in Form senkrechter Striche entlang der Abszisse, wobei jeder Strich für einen Wert steht. Um Bindungen in Form separater Striche in ihre Einzelwerte aufzulösen, sollte die Funktion mit `jitter()` gekoppelt werden. Diese Funktion verändert die Werte um einen kleinen zufälligen Betrag, wodurch sich die Lage der Striche horizontal leicht verschiebt. Dies führt zu dickeren Strichen als Repräsentation der Bindungen.

```
> rug(jitter(height))
```

Soll über das Histogramm zum Vergleich eine theoretisch vermutete Dichtefunktion gelegt werden, kann diese etwa mit `curve(..., add=TRUE)` hinzugefügt werden. Dabei muss entweder das Histogramm relative Häufigkeiten darstellen, oder aber die Skalierung der theoretischen Verteilungskurve angepasst werden.

¹⁵ `PlotFdist()` aus dem Paket `DescTools` stellt das Histogramm gemeinsam mit der kumulierten empirischen Häufigkeitsverteilung sowie einem boxplot in einem Diagramm dar. Für den Vergleich der Verteilungen einer Variablen in zwei Bedingungen zeigt `histbackback()` aus dem `Hmisc` Paket die zugehörigen Histogramme Rücken-an-Rücken angeordnet simultan in einem Diagramm.

¹⁶ Wird die Anzahl der Klassengrenzen genannt, behandelt R diesen Wert nur als Vorschlag, nicht als zwingend. Der in der Voreinstellung verwendete Algorithmus erzeugt häufig nur wenige Intervalle. Durch `breaks="FD"` wird ein anderer Algorithmus verwendet, der meist zu einer etwas größeren Anzahl führt.

```
# füge Dichtefunktion einer Normalverteilung hinzu
> curve(dnorm(x, mean(height), sd(height)), lwd=2, col="blue", add=TRUE)
```

Die Wahl der Intervallgrenzen hat einen starken Einfluss auf die Form von Histogrammen. Aufgrund dieser Abhängigkeit von einem willkürlich festzulegenden Parameter eignen sich Histogramme nicht immer gut, um sich einen Eindruck von der empirischen Verteilung einer Variable zu verschaffen.

Als Alternative lässt sich mit `density(<Vektor>)` auf Basis einer im Vektor gespeicherten Stichprobe von Werten die Dichtefunktion der zugehörigen Variable schätzen und grafisch darstellen.¹⁷ Die Berechnung der Schätzung kann über eine Reihe zusätzlicher Argumente kontrolliert werden, vgl. `?density`. Über `plot(density(<Vektor>))` wird die Schätzung der Dichtefunktion grafisch in einem separaten Diagramm veranschaulicht, während `lines(density(<Vektor>))` die geschätzte Dichtefunktion einem bereits geöffneten Diagramm hinzufügt. Bei einem Histogramm ist dabei darauf zu achten, für die Darstellung relativer Häufigkeiten das Argument `freq=FALSE` zu setzen.

```
> hist(height, freq=FALSE, xlim=fromTo, xlab="height [cm]",
+       main="Histogramm und Schätzung der Dichte")
> lines(density(height), lwd=2, col="blue")
> rug(jitter(height))
```

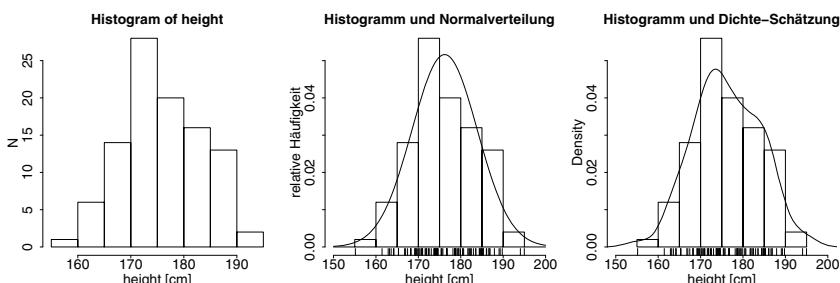


Abbildung 14.18: Histogramm, zusätzlich mit Einzelwerten und Normalverteilung bzw. mit Schätzung der Dichtefunktion

14.6.2 Stamm-Blatt-Diagramm

Das Stamm-Blatt-Diagramm mischt die Darstellung von Häufigkeiten einzelner Werte mit einem Histogramm. Seine Ausgabe erfolgt nicht in einem device, sondern in Textform auf der Konsole. Die Werte werden dafür zunächst wie bei einem Histogramm in disjunkte Intervalle eingeteilt, die dieselbe Breite besitzen. Diese Intervalle bilden den *Stamm* und werden durch die Ziffernfolge repräsentiert, mit der alle Werte im Intervall beginnen. Geht das erste Intervall etwa von 120 bis (ausschließlich) 130, ist der Wert des Stamms 12. Die *Blätter* werden dann jeweils durch jene Werte gebildet, die in dasselbe Intervall fallen und durch die auf den Stamm

¹⁷Zweidimensionale Dichten werden von `smoothScatter()` geschätzt und dargestellt.

folgende Ziffer repräsentiert. Dabei werden die Werte zunächst auf die Stelle gerundet, die auf den Stamm folgt. Insgesamt repräsentiert also jedes Blatt einen Wert der Stichprobe.

```
> stem(x=<Vektor>, scale=1, width=80, atom=1e-08)
```

Unter `x` wird der Datenvektor eingetragen. Mit `scale` kann die Anzahl der Intervalle in Form eines Skalierungsfaktors verändert werden. `width` legt mit Werten > 10 fest, wie viele Blätter maximal gezeigt werden, wobei diese Anzahl `width`-10 beträgt. Bei Werten für `width` ≤ 10 werden keine Blätter angezeigt – es wird nur vermerkt, ob mehr als `width` Werte im Intervall liegen und ggf. wie viele dies sind. Unter `atom` wird die Genauigkeit der Unterscheidung zwischen den einzelnen Werten definiert. Per Voreinstellung wird bis zur achten Nachkommastelle unterschieden.

```
> stem(rnorm(100, mean=175, sd=7))
The decimal point is 1 digit(s) to the right of the |
```

```
15 | 8
16 | 12344
16 | 5556678888999999
17 | 001111222222222334444
17 | 55555555666667777788889999999
18 | 00001223333444
18 | 5666899
```

Aus der Konstruktion des Diagramms folgt, dass die Kombination des Stammes mit einem zugehörigen Blatt einen Wert von `x` repräsentiert, weswegen sich alle Werte der Stichprobe aus dem Diagramm (bis auf die Rundung) rekonstruieren lassen. Ist der Stamm 15 und ein Blatt 8, wird etwa der Wert 158 dargestellt – in Abhängigkeit von der Lage der Dezimalstelle, über die in der Diagrammüberschrift informiert wird.

14.6.3 Boxplot

Ein boxplot (*box-whisker-plot*) stellt die Lage und Verteilung empirischer Daten durch die gleichzeitige Visualisierung verschiedener Kennwerte dar. Der Median wird dabei durch eine schwarze horizontale Linie innerhalb einer Box gekennzeichnet, deren untere Grenze sich auf Höhe des ersten und deren obere Grenze sich auf Höhe des dritten Quartils befindet.¹⁸ Innerhalb des so gebildeten Rechtecks liegen damit die mittleren 50% der Werte, seine Länge ist gleich dem Interquartilabstand. Jenseits der Box erstrecken sich nach oben und unten dünne Striche (*whiskers*), deren Enden jeweils den extremsten Wert angeben, der noch keinen Ausreißer darstellt. Als Ausreißer werden dabei in der Voreinstellung solche Werte betrachtet, die um mehr als das Anderthalbfache des Interquartilabstands unter oder über der Box liegen. Solche Ausreißer werden schließlich durch Kreise gekennzeichnet.

Boxplots sind u. a. dazu geeignet, Symmetrie bzw. Schiefe unimodaler Verteilungen zu beurteilen. Zudem lassen sich Lage und Verteilung einer Variable für mehrere Gruppen getrennt vergleichen,

¹⁸Liegen geradzahlig viele Werte vor, wird das Rechteck nach oben und unten nicht exakt durch die Quartile begrenzt, vgl. `?boxplot.stats`.

indem die zugehörigen boxplots nebeneinander in ein Diagramm gezeichnet werden (Abb. 14.19).

```
> boxplot(x=<Vektor>, range=<Zahl>, notch=FALSE, horizontal=FALSE)
```

Bei einem einzelnen boxplot wird unter `x` der Datenvektor eingegeben. Stattdessen kann für `x` auch eine Modellformel der Form `<Messwerte> ~ <Faktor>` übergeben werden, wobei `<Faktor>` dieselbe Länge wie der Vektor `<Messwerte>` besitzt und für jeden von dessen Werten codiert, zu welcher Bedingung er gehört. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Hiermit werden getrennt für die von `<Faktor>` definierten n Gruppen boxplots nebeneinander in einem Diagramm dargestellt, wobei als x -Koordinaten die Zahlen $1, \dots, n$ verwendet werden. Dasselbe Ergebnis lässt sich erzielen, indem an `x` eine Matrix übergeben wird, deren Spalten die Gruppen definieren, für die jeweils ein boxplot darzustellen ist. Über das Argument `range` wird die Definition eines Ausreißers als Vielfaches des Interquartilabstands kontrolliert. Das Argument `notch` bestimmt, ob ein gekerbter boxplot gezeichnet werden soll. Für horizontal verlaufende Boxen ist `horizontal=TRUE` zu setzen. Der auf der Konsole nicht sichtbare Rückgabewert enthält in Form einer Liste Angaben zu den dargestellten statistischen Kennwerten.

```
> Nj <- 40 # Gruppengröße
> P <- 3 # Anzahl Gruppen
> DV <- rnorm(P*Nj, mean=100, sd=15) # Messwerte

# Faktor Gruppenzugehörigkeit
> IV <- gl(P, Nj, labels=c("Control", "Group A", "Group B"))
> Mj <- tapply(DV, IV, FUN=mean) # Gruppenmittel
> boxplot(DV ~ IV, ylab="Score", col=c("red", "blue", "green"),
+           main="Boxplots der Scores in 3 Gruppen")

> points(1:P, Mj, pch=16, cex=2) # zeige Gruppenmittel
```

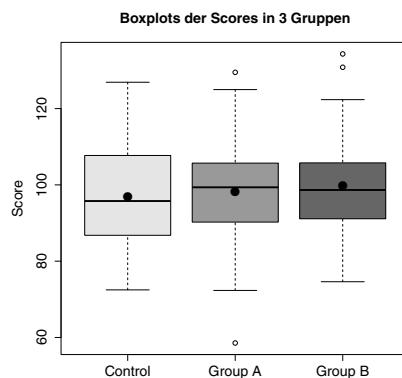


Abbildung 14.19: Boxplots getrennt nach Gruppen mit Mittelwerten

14.6.4 Stripchart

Ein mit `stripchart()` erstelltes eindimensionales Streudiagramm eignet sich zur Veranschaulichung der empirischen Verteilung quantitativer Variablen, wenn der Stichprobenumfang gering ist. Statt wie ein boxplot Daten summarisch anhand ihrer wichtigsten Verteilungsparameter zu illustrieren, stellt ein stripchart alle vorkommenden Werte selbst dar. Zu diesem Zweck wird jeder Einzelwert als Punkt auf einer horizontalen Achse repräsentiert, wobei der Wert die x -Koordinate des Punkts bestimmt (Abb. 14.20).

```
> stripchart(x=<Vektor>, method="overplot", vertical=FALSE, add=FALSE,
+             at=<Position>)
```

Die im Diagramm einzutragenden Daten werden in Form eines Vektors für x übergeben. In der Voreinstellung "overplot" bewirkt das Argument `method`, dass Bindungen durch dasselbe Symbol repräsentiert werden. Die so erstellte Grafik liefert damit keinen Aufschluss darüber, wie oft ein bestimmter Wert vorkommt. Um auch dies zu erreichen, gibt es zwei Methoden, die über das Argument `method` kontrolliert werden. Auf "jitter" gesetzt werden die Werte durch Symbole mit derselben x -Koordinate, aber einem zufälligen vertikalen Versatz dargestellt. Durch "stack" werden die Symbole eines mehrfach vorkommenden Wertes vertikal gestapelt. Um im Diagramm die Rolle von x - und y -Achse zu vertauschen, kann das Argument `vertical=TRUE` gesetzt werden.

Ein stripchart kann auch die Verteilung einer quantitativen Variable für mehrere Gruppen gleichzeitig veranschaulichen. Hierfür gibt es die Möglichkeit, mit `add=TRUE` das Ergebnis eines `stripchart()` Aufrufs einem schon bestehenden Diagramm hinzuzufügen – etwa einem boxplot zur gleichzeitigen Veranschaulichung der Rohdaten und wichtiger Kennwerte (Abb. 1.2). In diesem Fall kontrolliert das Argument `at` die vertikale Position der Achse, auf der die Symbole einzuziehen sind. Beim ersten Aufruf von `stripchart()` wird die Achse auf einer Höhe von 1 eingezeichnet.

Alternativ können die Daten auch als Modellformel $\langle \text{Messwerte} \rangle \sim \langle \text{Faktor} \rangle$ übergeben werden, wobei $\langle \text{Faktor} \rangle$ dieselbe Länge wie der Vektor $\langle \text{Messwerte} \rangle$ besitzt und für jeden von dessen Werten codiert, zu welcher Bedingung er gehört. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. In diesem Fall wird für jede Stufe von $\langle \text{Faktor} \rangle$ jeweils ein stripchart der zugehörigen Daten im Diagramm eingezeichnet, wobei unterschiedliche Faktorstufen durch vertikal getrennte Achsen kenntlich gemacht werden (Abb. 14.20).

```
> Nj    <- 25                                # Würfe pro Gruppe
> P     <- 4                                  # Anzahl Gruppen
> dice <- sample(1:6, P*Nj, replace=TRUE)      # Würfelwürfe
> IV   <- gl(P, Nj)                          # Gruppenzugehörigkeit
> stripchart(dice ~ IV, xlab="Augenzahl", ylab="Gruppe",
+             pch=1, col="blue", main="Würfelwürfe - 4 Gruppen",
+             sub="jitter-Methode", method="jitter")

> stripchart(dice ~ IV, xlab="Augenzahl", ylab="Gruppe",
+             pch=16, col="red", main="Würfelwürfe - 4 Gruppen",
+             sub="stack-Methode", method="stack")
```

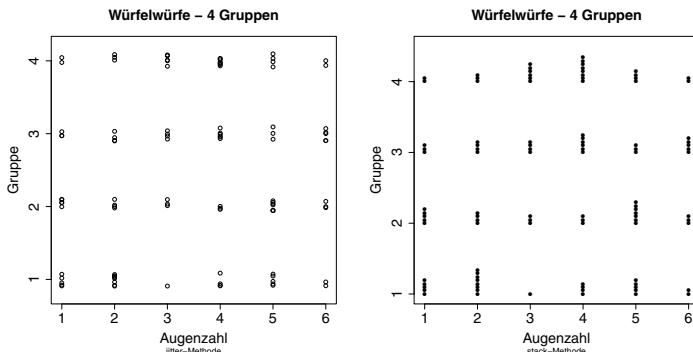


Abbildung 14.20: Stripcharts mit verschiedenen Methoden zur Darstellung einzelner Werte

14.6.5 Quantil-Quantil-Diagramm

Viele statistische Auswertungen setzen voraus, dass die zu analysierenden Variablen auf Ebene der Population eine bestimmte Verteilung aufweisen, etwa normalverteilt sind oder der Verteilung in einer anderen Population gleichen (vgl. Abschn. 10.1). Inwieweit die empirischen Werte mit dieser Annahme verträglich sind, kann mit einem Q-Q-Diagramm (Quantil-Quantil Darstellung) heuristisch abgeschätzt werden.

Vergleich zweier Stichproben

Wenn die Verteilung eines Merkmals in zwei Gruppen identisch ist, stimmen auch die Quantile überein (vgl. Abschn. 5.3.3). Auf empirischer Ebene lassen sich dafür die beobachteten Quantile von zwei Variablen zu denselben Wahrscheinlichkeiten in einem Diagramm gegeneinander auftragen. Sind ihre Quantile identisch, fallen die Punkte bei gleicher Achsenkalierung auf die Winkelhalbierende. Unterscheiden sich die Verteilungen lediglich durch ihre Skalierung, fallen die Punkte auf eine Gerade. Voneinander abweichende Verteilungen werden entsprechend durch Abweichungen von einer Referenzgerade deutlich. Ein solches Diagramm erstellt `qqplot(x=<Vektor>, y=<Vektor>)` (Abb. 14.21). Für x und y sind dafür die zu vergleichenden Variablen einzutragen, die auch unterschiedliche Länge haben können.

```
> DV1 <- rnorm(200)          # normalverteilte Messwerte simulieren
> DV2 <- rf(100, df1=3, df2=15)  # F-verteilte Werte simulieren, anderes N
> qqplot(DV1, DV2, xlab="Quantile N(0, 1)", ylab="Quantile F(3, 15)",
+         main="Quantile N(0, 1) vs. F(3, 15)-Verteilung")
```

Vergleich mit theoretischer Verteilung

Die empirischen Quantile von Daten einer Stichprobe können auch mit Quantilen verglichen werden, die sich aus der Annahme einer bestimmten Verteilung ergeben. Hierfür werden die

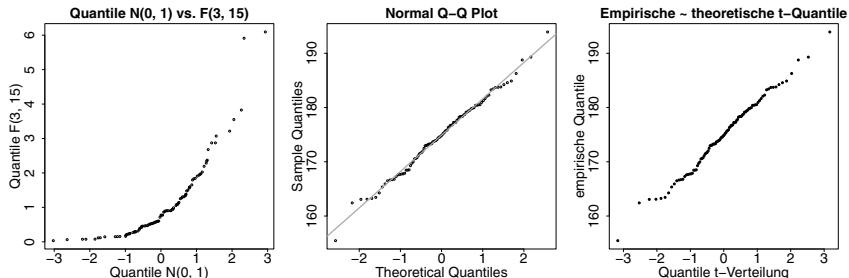


Abbildung 14.21: Quantil-Quantil Darstellung zum Vergleich von zwei Stichproben, zur Überprüfung von Normalverteiltheit und zum Vergleich mit t -Verteilung

erwarteten Quantile als x - und die tatsächlichen Quantile als y -Koordinaten von Punkten in einem Streudiagramm verwendet. Die empirischen Quantile sind einfach die sortierten Werte eines Vektors. Die Funktion `ppoints(Vektor)` bestimmt, welches der zu einem Quantil gehörende Wert der empirischen kumulierten Häufigkeitsverteilung als Schätzung der Verteilungsfunktion ist.¹⁹ Ihre Ausgabe kann dann als Argument für die Quantilfunktion einer infrage kommenden Verteilung verwendet werden, um die theoretischen Quantile zu erhalten (vgl. Abschn. 5.3.3). Quantile zu denselben Wahrscheinlichkeiten lassen sich dann mit `plot()` gegeneinander auftragen.

```
> cProb <- ppoints(height)           # kumulierte Wkt. für t-Quantile
> qTheo <- qt(cProb, df=10)         # t-Quantile
> qEmp <- sort(height)             # empirische Quantile

# Q-Q-Plot Vergleich empirische Quantile mit t-Quantilen
> plot(qTheo, qEmp, main="Empirische ~ theoretische t-Quantile",
+       xlab="Quantile t-Verteilung", ylab="empirische Quantile", pch=20)
```

Für den häufigen Spezialfall eines Vergleichs mit der Standardnormalverteilung existiert `qqnorm()`. Eine Referenzgerade wird durch `qqline()` in das Diagramm eingetragen (Abb. 14.21).

```
> qqnorm(y=<Vektor>, datax=FALSE)
> qqline(y=<Vektor>, datax=FALSE)
```

Mit `datax=FALSE` wird in der Voreinstellung festgelegt, dass die Quantile der empirischen Werte auf der y -Achse abgetragen werden. `datax` ist beim Aufruf von `qqnorm()` und `qqline()` auf denselben Wert zu setzen. Im Vergleich zu `qqplot()` entfällt hier die Angabe des ersten Datenvektors, da die x -Koordinaten der Punkte von den theoretisch erwarteten Quantilen gebildet werden.

```
> height <- rnorm(100, mean=175, sd=7)
> qqnorm(height)
> qqline(height, col="red", lwd=2)
```

¹⁹Dies sind die Mittelpunkte der Intervalle, die durch die empirischen kumulierten relativen Häufigkeiten gebildet werden (vgl. Abschn. 2.10.6).

14.6.6 Empirische kumulierte Häufigkeitsverteilung

Die kumulierte Häufigkeitsverteilung empirischer Daten lässt sich durch die `ecdf()` Funktion ermitteln, die ihrerseits eine Funktion erzeugt (vgl. Abschn. 2.10.6). Zur grafischen Darstellung wird diese neue Funktion an `plot()` übergeben (Abb. 14.22).

```
> vec <- round(rnorm(10), 1)
> Fn  <- ecdf(vec)
> plot(Fn, main="Empirische kumulierte Häufigkeitsverteilung")
> curve(pnorm, add=TRUE, col="gray", lwd=2)  # Vergleich mit Standard-NV
```

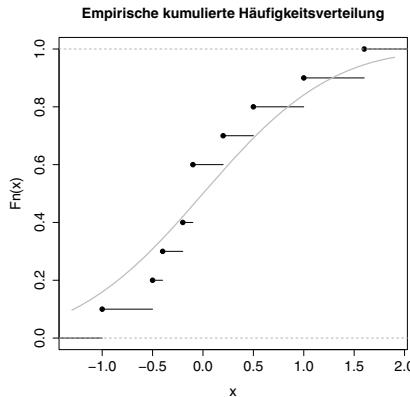


Abbildung 14.22: Vergleich von empirischen kumulierten Häufigkeiten mit der Verteilungsfunktion der Standardnormalverteilung

14.6.7 Kreisdiagramm

Das Kreis- oder auch Tortendiagramm ist eine weitere Möglichkeit, die Werte einer diskreten Variable grob zu veranschaulichen. Hier repräsentiert die Größe eines farblich hervorgehobenen Kreissektors (als stilisiertes Tortenstück) den Anteil des zugehörigen Wertes an der Summe aller Werte.²⁰ Da die korrekte Einschätzung von Flächeninhalten bzw. Sektorgrößen deutlich schwerer fällt als etwa der Vergleich von Linienlängen, sind Kreisdiagramme oft schlecht ablesbar und werden daher nicht häufig im wissenschaftlichen Kontext verwendet (Abb. 14.23).

```
> pie(x=<Vektor>, labels="<Namen>", col="<Farben>")
```

Für `x` ist ein Datenvektor mit nicht negativen Werten anzugeben. Sollen die einzelnen Sektoren mit einer Bezeichnung versehen werden, kann ein Vektor aus entsprechenden Zeichenketten für das Argument `labels` übergeben werden. Das Argument `col` kontrolliert die Farbe der Sektoren.

²⁰Für eine grafisch aufwendigere Darstellung von Kreisdiagrammen vgl. `pie3D()` aus dem `plotrix` Paket.

```
> dice <- sample(1:6, 100, replace=TRUE)      # Würfelwürfe
> dTab <- table(dice)                      # absolute Häufigkeiten
> pie(dTab, col=c("blue", "red", "yellow", "pink", "green", "orange"),
+      main="Relative Häufigkeiten beim Würfeln")
```

Um die Kreissektoren zu beschriften, wird hier `text()` verwendet. Die zur Plazierung notwendigen (x, y) -Koordinaten ergeben sich aus den relativen Häufigkeiten der Würfelergebnisse. Dabei ist dafür zu sorgen, dass die Beschriftung immer in der Mitte eines Sektors liegt. Die Kreismitte hat die Koordinaten $(0, 0)$, der Radius des Kreises beträgt 1. Die Beschriftungen liegen mit ihrem Mittelpunkt also auf einem Kreis mit Radius 0.5.

```
> dTabFreq <- prop.table(dTab)           # relative Häufigkeiten
> textRad <- 0.5                        # Radius für Beschriftungen
> angles <- dTabFreq * 2 * pi          # Sektorgrößen als Winkel
> csAngles <- cumsum(angles)           # Winkel der Sektorgrenzen
> csAngles <- csAngles - angles/2     # Winkel der Sektormitten
> textX <- textRad * cos(csAngles)    # x-Koordinaten für Text
> textY <- textRad * sin(csAngles)    # y-Koordinaten für Text
> text(x=textX, y=textY, labels=dTabFreq) # Sektorbeschriftungen
```

Relative Häufigkeiten beim Würfeln

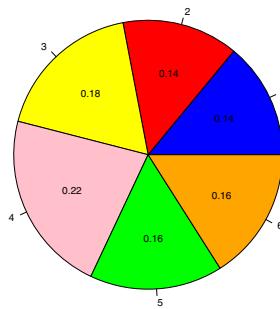


Abbildung 14.23: Kreisdiagramm als Mittel zur Darstellung von Kategorienhäufigkeiten

14.6.8 Gemeinsame Verteilung zweier Variablen

Die in Abschn. 14.2 vorgestellte `plot()` Funktion eignet sich dafür, die gemeinsame Verteilung von zwei Variablen in Form eines Streudiagramms zu untersuchen. Stammen die Daten aus verschiedenen Gruppen, kann die Gruppenzugehörigkeit über die Farbe oder den Typ der Datenpunktsymbole gekennzeichnet werden. Hierfür lässt sich die Eigenschaft von Faktoren ausnutzen, dass ihre Stufen intern über natürliche Zahlen repräsentiert sind, die sich mit `unclass()` ausgeben lassen und damit als Indizes dienen können (Abb. 14.24, vgl. Abschn. 14.8.3 für eine alternative Herangehensweise).

```

> N <- 200                                # Anzahl Personen
> P <- 2                                    # Anzahl Gruppen
> x <- rnorm(N, 100, 15)                   # Daten Variable 1
> y <- 0.5*x + rnorm(N, 0, 10)             # Daten Variable 2
> IV <- gl(P, N/P, labels=LETTERS[1:P])    # Gruppierungsfaktor

# gemeinsame Verteilung mit 2 Datenpunktsymbolen und Legende
> plot(x, y, pch=c(4, 16)[unclass(IV)], lwd=2,
+       col=c("black", "darkgray")[unclass(IV)],
+       main="Gemeinsame Verteilung getrennt nach Gruppen")

> legend(x="topleft", legend=c("Gruppe A", "Gruppe B"),
+          pch=c(4, 16), col=c("black", "darkgray"))

```

Weiterhin kann es hilfreich sein, ebenfalls die Streuungsellipse der gemeinsamen Verteilung einzuzeichnen, wie sie durch die Eigenwerte und Eigenvektoren der Kovarianzmatrix der Variablen definiert wird (vgl. Abschn. 12.1.5, 12.2). Die Ellipse hat ihren Mittelpunkt im Zentroid der Daten, ihre Hauptachsen sind durch die Eigenvektoren gegeben, wobei die Länge jeder Halbachse gleich der Wurzel aus dem zugehörigen Eigenwert ist (Abb. 14.24).²¹

```

> mat     <- cbind(x, y)                  # Datenmatrix
> ctr     <- colMeans(mat)                # Zentroid
> eigVal <- eigen(cov(mat))$values      # Eigenwerte
> eigVec <- eigen(cov(mat))$vectors     # Eigenvektoren

# mit der Wurzel der Eigenwerte skalierte Eigenvektoren
> eigScl <- eigVec %*% diag(sqrt(eigVal))

# Hauptachsen der Ellipse: Zentroid +/- skalierte Eigenvektoren
> xMat <- rbind(ctr[1] + eigScl[1, ], ctr[1] - eigScl[1, ])
> yMat <- rbind(ctr[2] + eigScl[2, ], ctr[2] - eigScl[2, ])

# Koordinaten der Ellipse: zunächst unrotiert
> angles <- seq(0, 2*pi, length.out=200)
> ellBase <- cbind(sqrt(eigVal[1])*cos(angles),
+                     sqrt(eigVal[2])*sin(angles))

# rotiere mit Matrix der standardisierten Eigenvektoren
> ellRot <- eigVec %*% t(ellBase)

# gemeinsame Verteilung der Variablen darstellen
> plot(mat, xlab="x", ylab="y", asp=1,
+       main="Gemeinsame Verteilung zweier Variablen")

# ins Zentroid verschobene Streuungsellipse einzeichnen
> polygon((ellRot+ctr)[1, ], (ellRot+ctr)[2, ], lwd=2, border="blue")

```

²¹Für Konfidenzellipsen im inferenzstatistischen Sinn vgl. `confidenceEllipse()` aus dem `car` Paket.

```
# Hauptachsen einzeichnen
> matlines(xMat, yMat, lty=1, lwd=2, col="green")

# Zentroid markieren
> points(ctr[1], ctr[2], pch=4, col="red", cex.lab=1.5, lwd=3)
```

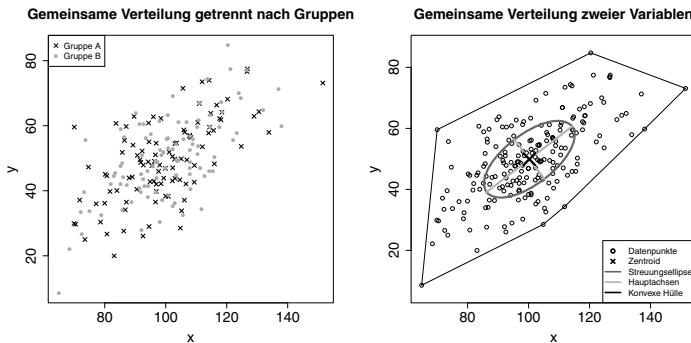


Abbildung 14.24: Gemeinsame Verteilung zweier Variablen getrennt nach Gruppen. Gesamtdaten mit Streuungsellipse, deren Hauptachsen und der konvexen Hülle

Die konvexe Hülle der gemeinsamen Verteilung zweier Variablen ist definiert als kleinstes konvexas Polygon, in dem, bzw. auf dessen Rand alle Datenpunkte liegen. Ihr mit `chull()` ermittelbarer Rand markiert die extremsten Datenpunkte der Verteilung und kann damit bei der Identifikation von Ausreißern hilfreich sein (Abb. 14.24).

```
> chull(x=<Vektor>, y=<Vektor>)
```

Unter `x` und `y` sind die Ausprägungen der ersten und zweiten Variable jeweils als Vektor einzutragen. Wird nur ein Vektor angegeben, werden seine Elemente als `y`-Koordinaten interpretiert und die `x`-Koordinate jedes Datenpunkts gleich dem Index des zugehörigen Vektorelements gesetzt. Die Ausgabe liefert im Uhrzeigersinn die Indizes der Ecken des Polygons.

```
> hullIdx <- chull(x, y) # Indizes Ecken konvexe Hülle
> polygon(x[hullIdx], y[hullIdx]) # konvexe Hülle einzeichnen
> legend(x="bottomright", legend=c("Datenpunkte", "Zentroid",
+ "Streuungsellipse", "Hauptachsen", "konvexe Hülle"),
+ pch=c(1, 4, NA, NA, NA), lty=c(NA, NA, 1, 1, 1),
+ col=c("black", "red", "blue", "green", "black"))
```

Sind nur wenige Wertepaare möglich und deswegen Bindungen in den Daten vorhanden, würden mehrere gleiche Wertepaare durch dasselbe Symbol im Diagramm dargestellt. Sollen dagegen unter Verzicht auf die präzise Positionierung ebenso viele Symbole wie Wertepaare angezeigt werden, kann dies mit `jitter(<Variable>)` erreicht werden. Diese Funktion ändert die Werte

der Variable um einen kleinen zufälligen Betrag und bewirkt dadurch beim Zeichnen jedes Datenpunkts einen zufälligen Versatz entlang der zugehörigen Achse (Abb. 14.25).²²

```
> vec1 <- sample(1:10, 100, replace=TRUE)
> vec2 <- sample(1:10, 100, replace=TRUE)
> plot(vec2 ~ vec1, main="Punktwolke")           # Datenpunkte ohne jitter()

# Datenpunkte mit zufälligem Versatz in Richtung der y-Achse durch jitter()
> plot(jitter(vec2) ~ vec1, main="Punktwolke mit jitter")
```

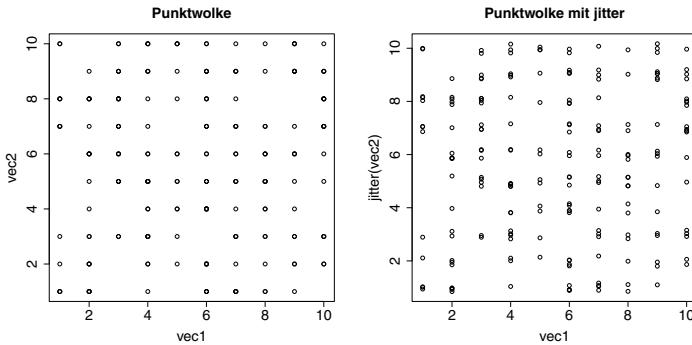


Abbildung 14.25: Streudiagramm von Daten mit Bindungen ohne und mit Anwendung von `jitter()`

Bei sehr vielen darzustellenden Punkten taucht mitunter das Problem auf, dass Datenpunkt-symbole einander überdecken und einzelne Werte nicht mehr identifizierbar sind. Indem als Datenpunktssymbol mit `pch=". "` der Punkt gewählt wird, lässt sich dies bis zu einem gewissen Grad verhindern. Auch der Einsatz von simulierter Transparenz für die Datenpunktssymbole kann einzelne Werte identifizierbar halten: Bei einer sehr durchlässig gewählten Farbe erzeugt ein einzelner Datenpunkt nur einen schwachen Abdruck, während viele aufeinander liegende Punkte die Farbe an dieser Stelle des Diagramms immer gesättigter machen (vgl. Abschn. 14.3.2, Fußnote 6 und Abb. 12.2).

Alternativ lässt sich mit `hexbin()` aus dem gleichnamigen Paket (Carr, Lewin-Koh & Maechler, 2014) ein Diagramm erstellen, das die Diagrammfläche in hexagonale Regionen einteilt und die Dichte der Datenpunkte in jeder Region ähnlich einem Höhenlinien-Diagramm (vgl. Abschn. 14.8.1) farblich repräsentiert. Auch `smoothScatter()` codiert die Dichte von Datenpunkten über farblich abgesetzte Diagrammregionen, verwendet dabei jedoch einen 2D-Kerndichteschätzer zur Glättung der Regionengrenzen.

²² Als Alternative zu diesem Vorgehen kommt `sunflowerplot()` in Betracht.

14.7 Daten interpolieren und fitten

R verfügt über verschiedene vorbereitete Möglichkeiten, zwischen gegebenen Datenpunkten im zweidimensionalen Raum zu interpolieren und Daten zu fitten. Hierbei werden Datenpunkte generiert, die horizontal und vertikal zwischen den bestehenden plaziert sind und diese i.S. einer zugrundeliegenden Funktion ergänzen.²³

14.7.1 Lineare Interpolation und LOESS-Glätter

`approx()` interpoliert linear zwischen Datenpunkten.

```
> approx(x=<x-Koordinaten>, y=<y-Koordinaten>,
+         method=<Interpolationsmethode>", n=<Anzahl>)
```

Die (x, y) -Koordinaten der Datenpunkte sind für x und y in Form von Vektoren derselben Länge zu übergeben. Mit dem Argument `method` wird kontrolliert, wie die Interpolation erfolgt – `"linear"` bewirkt eine lineare Interpolation, wie sie den mit `plot(..., type="l")` erzeugten Liniensegmenten entspricht. Mit `method="constant"` erhalten alle interpolierten Punkte die y -Koordinate des in horizontaler Richtung vorangehenden Datenpunkts. Das Ergebnis ähnelt dann jenem von `plot(..., type="s")`. Wie viele Punkte hinzugefügt werden sollen, kann über das Argument `n` festgelegt werden. Die Ausgabe von `approx()` besteht aus einer Liste mit den Komponenten x und y , den Vektoren der (x, y) -Koordinaten der interpolierten Punkte. Die Liste kann direkt etwa an `points()` übergeben werden.²⁴

```
> xOne      <- 1:9                                # x-Koord.
> yOne      <- rnorm(9)                            # y-Koord.
> ptsLin    <- approx(xOne, yOne, method="linear") # linear
> ptsConst <- approx(xOne, yOne, method="constant")# konstant

# linkes Diagramm
> plot(xOne, yOne, pch=19, cex=2, main="Datenpunkte interpolieren")
> points(ptsLin, pch=16, col="red", lwd=1.5)
> points(ptsConst, pch=22, col="blue", lwd=1.5)
> legend(x="bottomleft", c("Daten", "linear", "konstant"),
+         pch=c(19, 16, 22), col=c("black", "red", "blue"))
```

Um (x, y) -Streudiagramme durch glatte Kurven zu approximieren, existieren verschiedene Glätter, die auf lokal gewichteten Polynomen basieren – darunter die in `lowess()` und `loess.smooth()` implementierten (Abb. 14.26).²⁵

```
> lowess(x=<x-Koordinaten>, y=<y-Koordinaten>, f=<Spannweite>)
> loess.smooth(x=<x-Koordinaten>, y=<y-Koordinaten>, span=<Spannweite>)
```

²³Für Möglichkeiten zum Anpassen verschiedener Funktionsfamilien vgl. `fitdistr()` aus dem `MASS` Paket.

²⁴`approxfun()` gibt stattdessen eine Funktion zurück, die x -Koordinaten akzeptiert und y -Koordinaten ausgibt, so dass die ursprünglichen Daten interpoliert werden. `approxfun()` besitzt dieselben Argumente wie `approx()` bis auf die nicht benötigte Angabe `n`.

²⁵Weitere Glätter lassen sich mit `supsmu()` und `smooth()` anwenden.

Die (x, y) -Koordinaten der Datenpunkte sind für `x` und `y` in Form von Vektoren derselben Länge zu übergeben. Das Argument `f` von `lowess()` gibt ebenso wie das Argument `span` von `loess.smooth()` die Spannweite des Glätters als Anteil der Punkte an, deren Position in jedem geglätteten Wert berücksichtigt wird. Größere Anteile bewirken dabei glattere Interpolationen (Voreinstellung ist $\frac{2}{3}$). Beide Funktionen geben eine Liste mit den Komponenten `x` und `y` zurück, den Vektoren der (x, y) -Koordinaten der interpolierten Punkte.

```
# mittleres Diagramm
> xTwo <- rnorm(100)                                     # x-Koordinaten
> yTwo <- 0.4 * xTwo + rnorm(100, 0, 1)                  # y-Koordinaten
> ptsL1 <- loess.smooth(xTwo, yTwo, span=1/3)            # weniger glatt
> ptsL2 <- loess.smooth(xTwo, yTwo, span=2/3)            # glatter
> plot(xTwo, yTwo, xlab=NA, ylab=NA, pch=16,
+       main="Geglättetes Streudiagramm")

> lines(ptsL1, lwd=2, col="red")                           # Loess-Glättter einzeichnen
> lines(ptsL2, lwd=2, col="blue")
```

14.7.2 Splines

Splines sind parametrisierte Kurven, die sich dafür eignen, zwischen Werten glatt zu interpolieren bzw. Daten durch glatte Kurven zu approximieren, wenn keine theoretisch motivierte Funktion bekannt ist. Kubische splines können mit `spline()` und `smooth.spline()` erzeugt werden: Während die von `spline()` ermittelten Punkte dabei auf einer Kurve durch alle vorhandenen Datenpunkte liegen, ist dies für die von `smooth.spline()` berechnete Kurve nicht der Fall. Hier lässt sich die Kurve über das Argument `spar` (smoothing parameter) in ihrer Glattheit, d.h. in dem Ausmaß ihrer Variation kontrollieren. Dadurch wird auch bestimmt, wie nah sie an den Datenpunkten liegt.

`spline()` gibt eine Liste mit den (x, y) -Koordinaten in den Komponenten `x` und `y` aus, die direkt in Funktionen wie `lines()` verwendet werden kann.²⁶ Das Ergebnis von `smooth.spline()` ist ein Objekt, mit dem über `predict()` die interpolierten `y`-Koordinaten für neue `x`-Koordinaten erzeugt werden können.

`xspline()` stellt weitere splines bereit, mit denen nicht nur Funktionswerte interpoliert, sondern allgemein über sog. *Kontrollpunkte* beliebige, auch geschlossene und sich überschneidende Formen angenähert werden können (Abb. 14.26). Sofern gewünscht zeichnet `xspline()` das Ergebnis direkt in ein Diagramm ein.²⁷

```
> ord <- order(xTwo)                                     # für geordnete x-Koordinaten
> idx <- seq(8, 88, by=20)                             # wähle 4 Kontrollpunkte
> xspline(xTwo[ord][idx], yTwo[ord][idx], c(1, -1, -1, 1, 1),
+           border="darkgreen", lwd=2, open=FALSE)
```

²⁶ `splinefun()` gibt stattdessen eine Funktion zurück, die `x`-Koordinaten akzeptiert und `y`-Koordinaten ausgibt, so dass die ursprünglichen Daten interpoliert werden. `splinefun()` besitzt dieselben Argumente wie `spline()` bis auf die nicht benötigte Angabe `n`.

²⁷ Für weitere Spline-Typen vgl. `help(package="splines")`.

```

> legend(x="topleft", c("Daten", "Loess span 1/3", "Loess span 2/3",
+           "xspline"), pch=c(19, NA, NA, NA), lty=c(NA, 1, 1, 1),
+           col=c("black", "red", "blue", "darkgreen"))

# rechtes Diagramm
> plot(xOne, yOne, pch=19, cex=1.5, main="Splines")      # Daten
> ptsSpline <- spline(xOne, yOne, n=201)                 # kubischer spline

# smoothing splines unterschiedlicher Glattheit
> smSpline1 <- smooth.spline(xOne, yOne, spar=0.25)     # recht variabel
> smSpline2 <- smooth.spline(xOne, yOne, spar=0.35)     # mittel variabel
> smSpline3 <- smooth.spline(xOne, yOne, spar=0.45)     # recht glatt

# neue x-Koordinaten, auf die Splines angewendet werden
> ptsX       <- seq(1, 9, length.out=201)
> ptsSmSpl1 <- predict(smSpline1, ptsX)
> ptsSmSpl2 <- predict(smSpline2, ptsX)
> ptsSmSpl3 <- predict(smSpline3, ptsX)

# Linien einzeichnen
> lines(ptsSpline, col="darkgray", lwd=2)
> matlines(x=ptsX, y=cbind(ptsSmSpl1$y, ptsSmSpl2$y, ptsSmSpl3$y),
+           col=c("blue", "green", "orange"), lty=1, lwd=2)

> legend(x="bottomleft", c("data", "spline", "spar=0.3", "spar=0.4",
+           "spar=0.5"), pch=c(19, NA, NA, NA, NA), lty=c(NA, 1, 1, 1, 1),
+           col=c("black", "darkgray", "blue", "green", "orange"))

```

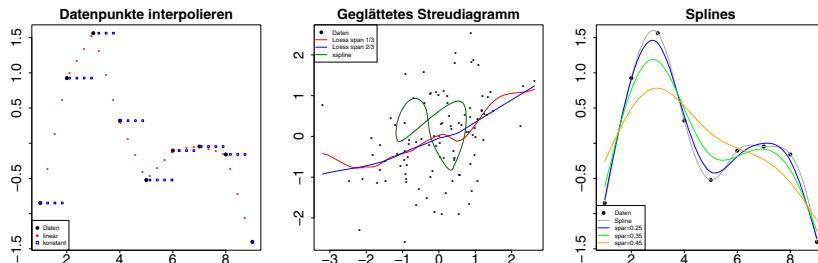


Abbildung 14.26: Lineare und konstante Interpolation von Daten, polynomiale Glätter sowie unterschiedlich glatte splines

14.8 Multivariate Daten visualisieren

In unterschiedlichen Situationen kann es erstrebenswert sein, multivariate Daten grafisch darzustellen: So Messwerte in ihrer Ausprägung von mehr als einer numerischen Variable abhängen,

etwa im Kontext einer multiplen linearen Regression. Oder aber mehrere Variablen werden gleichzeitig erhoben, um ihre gemeinsame Verteilung zu analysieren. Schließlich hängt etwa im Rahmen mehrfaktorieller Varianzanalysen eine Variable von der Kombination verschiedener qualitativer Faktoren ab. Die Visualisierung solcher Daten muss dann alle beteiligten Komponenten berücksichtigen. Da Diagramme nur zweidimensional sein können, ergibt sich dabei das Problem, dass die räumliche Lage eines Datenpunkts in der Plot-Region nicht mehr als zwei Komponenten repräsentieren kann.

Für dreidimensionale Daten existieren als Ausweg verschiedene Diagrammtypen, die sich darin unterscheiden, auf welche Weise die dritte Komponente grafisch codiert wird:²⁸ So wird versucht, einen räumlichen Tiefeneindruck zu erzeugen und die dritte Komponente als z -Koordinate, d. h. als Höhe eines Datenpunkts über einer Ebene zu repräsentieren (dreidimensionale Streudiagramme und Gitterflächen). Oder die dritte Komponente wird nicht räumlich, sondern farblich bzw. durch andere grafische Unterscheidungsmerkmale symbolisiert, etwa durch Höhenlinien oder die Größe der Datenpunktssymbole. Bei mehr als drei Variablen kann auf die direkte Veranschaulichung aller Komponenten zugunsten einer aufgeteilten Grafik verzichtet werden, in der eine Serie uni- oder bivariater Diagramme nebeneinander für alle Stufen bzw. Stufenkombinationen weiterer Variablen angeordnet ist.

14.8.1 Höhenlinien und variable Datenpunktssymbole

Den Diagrammtypen, die Höhenlinien oder Gitter zur Visualisierung der z -Koordinate verwenden, ist die Art der Angabe von Koordinaten gemein. Sie benötigen zum einen zwei Vektoren, die die Werte auf der x - und y -Achse festlegen. Als drittes Argument erwarten die Funktionen eine Matrix, die Werte für jede Kombination der übergebenen (x, y) -Koordinaten enthält und deswegen so viele Zeilen wie x - und so viele Spalten wie y -Koordinaten besitzt. Die Werte dieser Matrix definieren die z -Koordinate als dritte Komponente für jedes (x, y) -Koordinatenpaar.

Höhenlinien symbolisieren die z -Koordinate wie topografische Karten durch die Zugehörigkeit eines Punkts zu einer Region der Diagrammfläche, die durch eine geschlossene Höhenlinie definiert wird. Dieses Vorgehen ist mit einer Vergrößerung der Daten verbunden, da Wertebereiche in Kategorien zusammengefasst werden.

```
> contour(x=<x-Koordinaten>, y=<y-Koordinaten>, z=<z-Koordinaten>,
+           nlevels=<Anzahl Höhenlinien>, levels=<Kategorien>,
+           labels=<Namen>, drawlabels=TRUE)
```

Für x und y muss jeweils ein Vektor mit den x - bzw. y -Koordinaten der Datenpunkte übergeben werden. Die Matrix z definiert die z -Koordinaten in der o. g. Form. Welche Kategorien Verwendung finden, kann über die Argumente `levels` und `nlevels` kontrolliert werden. Für `levels` ist ein Vektor aus Kategorienbezeichnungen zu übergeben. In diesem Fall ist die Verwendung von `nlevels` nicht mehr notwendig, da sich die Zahl der Kategorien aus der Länge von `levels` ergibt. Andernfalls ist für `nlevels` die gewünschte Anzahl an Kategorien zu nennen. Das Argument `drawlabels` bestimmt, ob die Kategorienbezeichnungen im Diagramm eingetragen werden.

²⁸Ein Ansatz, höherdimensionale Daten durch Sequenzen von Projektionen auf niedrigdimensionale Räume zu visualisieren, ist im Paket `tourr` (Wickham, Cook, Hofmann & Buja, 2011) umgesetzt.

Im folgenden Beispiel soll die Dichtefunktion von zwei gemeinsam normalverteilten Variablen mit positiver Korrelation dargestellt werden (Abb. 14.27). Um die Dichte zu berechnen, wird zunächst eine eigene Funktion definiert (vgl. Abschn. 15.2), die `dmvnorm()` aus dem `mvtnorm` Paket für ein (x, y) -Koordinatenpaar aufruft. Mit `outer()` lässt sich die Funktion dann auf alle Paare von (x, y) -Koordinaten anwenden.

```
> mu      <- c(1, 3)                  # Erwartungswerte Normalverteilung
> sigma   <- matrix(c(1, 0.6, 0.6, 1), nrow=2)          # Kovarianzmatrix
> rng     <- 2.5                      # Wertebereich in Std.-Abw.
> N       <- 50                       # Anzahl der x- und der y-Koordinaten

# x- und y-Koordinaten
> X <- seq(from=mu[1] - rng*sigma[1, 1],
+           to=mu[1] + rng*sigma[1, 1], length.out=N)

> Y <- seq(from=mu[2] - rng*sigma[2, 2],
+           to=mu[2] + rng*sigma[2, 2], length.out=N)

# Funktion, um Dichte z für gegebenes (x, y)-Paar zu erzeugen
> library(mvtnorm)                   # für dmvnorm()
> genZ <- function(x, y) { dmvnorm(cbind(x, y), mu, sigma) }

# Dichte z für alle (x, y)-Koordinatenpaare berechnen
> matZ <- outer(X, Y, "genZ")
> contour(X, Y, matZ, main="Höhenlinien für 2D-NV Dichte")
```

Die durch die Höhenlinien definierten Regionen können mit `filled.contour()` auch eingefärbt werden. Dabei stammen die Farben aus einem Farbverlauf, dessen Zuordnung zu Kategorien auf der rechten Seite des Diagramms in einer Legende erläutert wird (Abb. 14.27). Der Aufruf gleicht dem für `contour()` stark, jedoch kann über das zusätzliche Argument `color.palette` eine eigene Farbpalette spezifiziert werden.

```
> filled.contour(X, Y, matZ, main="Farbige Höhenlinien")
```

Mit `symbols()` lassen sich zwei – typischerweise quantitative – Variablen durch die räumliche Lage von Datenpunktssymbolen gemeinsam mit weiteren Variablen durch die Gestaltung dieser Symbole grafisch repräsentieren (Abb. 14.27).²⁹

```
> symbols(x=<Vektor>, y=<Vektor>, add=FALSE, circles=<Vektor>,
+           boxplots=<Nx5 Matrix>, inch=TRUE, fg=<Farben>, bg=<Farben>)
```

Für `x` und `y` muss jeweils ein Vektor mit den x - bzw. y -Koordinaten der Datenpunkte genannt werden, alternativ ist auch eine Modellformel der Form $\langle y \rangle \sim \langle x \rangle$ möglich. Welche Datenpunktssymbole an diesen Koordinaten erscheinen, richtet sich danach, für welches weitere Argument Daten übergeben werden: `circles` erwartet einen Vektor derselben Länge wie `x` und `y` mit positiven Werten für die Größe der als Datenpunktssymbol dienenden Kreise. Der größte Wert entspricht in der Voreinstellung `inch=TRUE` einer Symbolgröße von 1 in, die Größe der übrigen

²⁹Ähnliche Abbildungen erzeugen `stars()` und `sunflowerplot()`.

Symbole richtet sich nach dem Verhältnis der zugehörigen Werte zum Maximum. Wird an `inch` stattdessen ein numerischer Wert übergeben, definiert er die Maximalgröße der Symbole. Um an jedem Koordinatenpaar einen Boxplot zu zeichnen, muss an `boxplots` eine Matrix mit 5 Spalten und so vielen Zeilen übergeben werden, wie `x` und `y` jeweils Elemente besitzen. Die Werte in den Spalten stehen dabei für die Breite und Höhe der Boxen, für die Länge der unteren und oberen Striche (whiskers) sowie für den Median. Für weitere Symbole vgl. `?symbols`. Die Farben der Symbole legt `fg`, die der von ihnen umschlossenen Flächen `bg` fest.

Das Beispiel soll anhand eines (sicher unrealistischen) Modells den Zusammenhang zwischen den negativ korrelierten Prädiktoren Alter und Sport (in Minuten pro Woche) und dem Körpergewicht als gemessene Variable simulieren.

```
> N      <- 10
> age    <- rnorm(N, 30, 8)
> sport   <- abs(-0.25*age + rnorm(N, 60, 30))
> weight <- -0.3*age -0.4*sport + 100 + rnorm(N, 0, 3)

# reskaliere Werte für das Körpergewicht auf das Intervall [0.2, 1]
> wScale <- (weight-min(weight)) * (0.8 / abs(diff(range(weight)))) + 0.2
> symbols(age, sport, circles=wScale, inch=0.6, fg=NULL, bg=rainbow(N),
+           main="Gewicht vs. Alter und Sport")
```

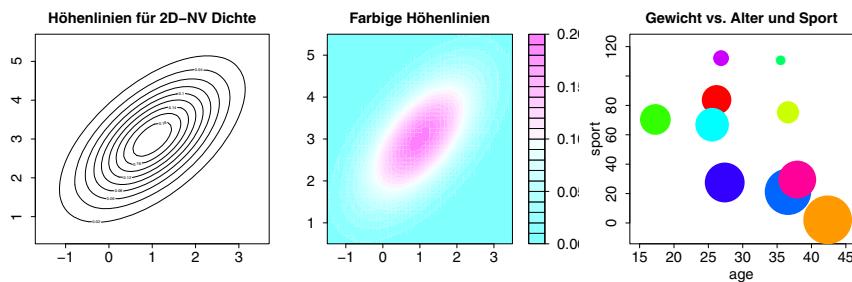


Abbildung 14.27: Visualisierungsmöglichkeiten für dreidimensionale Daten: Höhenlinien und variable Datenpunktssymbole

14.8.2 Dreidimensionale Gitter und Streudiagramme

`persp()` erzeugt Diagramme mit Tiefeneindruck, in der die Datenpunkte zu einer Gitterfläche verbunden sind. Der Augpunkt, d. h. die Perspektive, aus der die simulierte dreidimensionale Szene gezeigt wird, ist frei wählbar (Abb. 14.28).

```
> persp(x=<x-Koordinaten>, y=<y-Koordinaten>, z=<z-Koordinaten>,
+        theta=0, phi=15, r=sqrt(3))
```

Die Argumente `x`, `y` und `z` haben dieselbe Bedeutung wie in Höhenlinien-Diagrammen. Die Argumente `theta`, `phi` und `r` bestimmen die Blickrichtung auf das Diagramm in Form von

Polarkoordinaten, die innerhalb einer gedachten Kugel mit dem Ursprung des Koordinatensystems im Zentrum definiert sind. Dabei bezeichnet `theta` den Azimuth (Längengrad) und `phi` die Höhe über dem Äquator (Breitengrad, Elevation). Die Entfernung zum Diagramm als Kugelradius kontrolliert `r`. Die Funktion verfügt über weitere Argumente, u. a. zur Kontrolle der perspektivischen Verzerrung und zur räumlichen Kompression der `z`-Achse.

```
> persp(X, Y, matZ, xlab="x", ylab="y", zlab="Dichte", theta=5,
+       phi=35, main="Dichte einer 2D Normalverteilung")
```

Mit Funktionen aus dem Paket `rgl` (D. Adler & Murdoch, 2014) erstellte Diagramme – etwa `plot3d()`, `hist3d()` oder `persp3d()` als Pendants zu den konventionellen Funktionen `plot()`, `hist()` und `persp()`, erlauben durch den Einsatz der Grafikbibliothek OpenGL eine interaktive Manipulation:³⁰ Durch Anklicken der Diagrammfläche lässt sich die Perspektive auf das Diagramm beliebig ändern, indem die linke Maustaste gedrückt gehalten und die Maus bewegt wird (Abb. 14.28).

```
> plot3d(x=<x-Koordinaten>, y=<y-Koordinaten>, z=<z-Koordinaten>)
```

Im Unterschied zu `contour()` und `persp()` müssen die (x, y, z) -Koordinaten der Punkte hier in Form dreier Vektoren gleicher Länge an die Argumente `x`, `y` und `z` übergeben werden. Das `rgl` Paket bringt eigene Funktionen zum Einfügen aller in Abschn. 14.5 für zweidimensionale Diagramme beschriebenen Grafikelemente mit, vgl. `help(package="rgl")`.

```
> vecX <- rep(seq(-10, 10, length.out=10), times=10)
> vecY <- rep(seq(-10, 10, length.out=10), each=10)
> vecZ <- vecX*vecY
> library(rgl)                                     # für plot3d()
> plot3d(vecX, vecY, vecZ, main="3D Scatterplot",
+         col="blue", type="h", aspect=TRUE)

> spheres3d(vecX, vecY, vecZ, col="red", radius=2)
> grid3d(c("x", "y+", "z"))
```

Das `rgl` Paket enthält viele Zusatzfunktionen, mit denen der dreidimensionale Eindruck eines Diagramms über die Plazierung von Lichtquellen und die Manipulation von simulierten Oberflächenbeschaffenheiten gesteigert werden kann. Durch `demo(rgl)` sowie insbesondere `example(persp3d)` erhält man einen Überblick über die zahlreichen Gestaltungsmöglichkeiten, die das Paket eröffnet.

14.8.3 Bedingte Diagramme für mehrere Gruppen mit `ggplot2`

Multivariate Daten, deren Variablen teilweise kategorial sind, lassen sich durch bedingte Diagramme visualisieren: Dafür wird die gesamte Diagrammfläche in *Facetten (panels)* unterteilt

³⁰ Auch die Pakete `rggobi` (Temple Lang, Swayne, Wickham & Lawrence, 2014) und `playwith` (Andrews, 2012) stellen Funktionen bereit, die Grafiken interaktiv in Echtzeit verändern, sie etwa an geänderte Ursprungsdaten anpassen. Sehr dynamisch ist derzeit die Entwicklung von Paketen, die interaktive Diagramme für Webseiten mit Hilfe von JavaScript-Bibliotheken implementieren. Dazu zählen `googleVis` (Gesmann & de Castillo, 2011), `ggvis` (RStudio Inc, 2014a) und `rCharts` (Vaidyanathan, 2013).

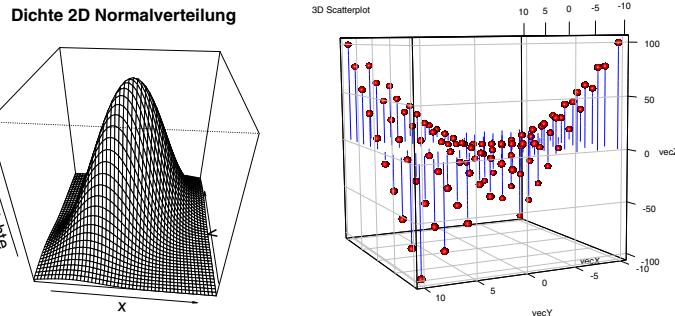


Abbildung 14.28: Visualisierungsmöglichkeiten für dreidimensionale Daten: Gitter und interaktives Streudiagramm

und in den Facetten getrennt für jede Stufe einer kategorialen Variable derselbe Diagrammtyp – etwa ein Streudiagramm – dargestellt. Bei mehreren Faktoren ist dies für jede Kombination von Stufen verschiedener Faktoren möglich.

Nach Gruppenzugehörigkeit getrennte Diagramme lassen sich besonders einfach mit Funktionen aus dem Paket `ggplot2` darstellen, das Ideen der *grammar of graphics* umsetzt (L. Wilkinson, 2005).³¹ Die so erzeugten Diagramme sind oft ästhetisch ansprechender als die Diagramme des Basisumfangs von R, allerdings weicht die Verwendung grundsätzlich von den bisher vorgestellten Funktionen ab. Daher sollen hier nur wenige Beispiele vorgestellt werden, für Details vgl. Wickham (2009) oder Chang (2013), dessen Webseite viele Varianten erklärt. Grundlage für alle mit `ggplot2` erzeugten Diagrammtypen ist die Funktion `ggplot()`.

```
> ggplot(<Datensatz>, aes(x=<x-Koord.>, y=<y-Koord.>,
+                           color=<Faktor>,
+                           fill=<Faktor>,
+                           shape=<Faktor>,
+                           group=<Faktor>))
```

Als erstes Argument wird der Datensatz erwartet, aus dem die im Diagramm dargestellten Variablen stammen. Bei Daten aus Messwiederholungen sollte der Datensatz im Long-Format vorliegen (vgl. Abschn. 3.3.9). Das zweite Argument bestimmt mit der Funktion `aes()`, welche Werte für die *x*- und *y*-Koordinaten herangezogen werden. Zusätzliche Argumente für `aes()` kontrollieren ästhetische Eigenschaften der Diagrammelemente, die dann automatisch in der Legende erklärt werden. Unter anderem sind dies folgende:

- **color:** Ein übergebener Faktor sorgt dafür, dass die Datenpunkte getrennt nach Faktorstufe eingefärbt sind. Welche Farbe welcher Gruppe zugeordnet ist, wählt `ggplot()` automatisch selbst.
- **fill:** Bei flächigen Diagrammelementen – etwa Säulen – kontrolliert der übergebene Faktor analog zu `color` deren Farbe.

³¹ `ggplot2` und `lattice` setzen nicht auf dem grafischen Basisystem von R auf. Damit ergeben sich bzgl. der Feinkontrolle von Grafiken einige Abweichungen zu den Darstellungen in Abschn. 14.3 und 14.5 (Murrell, 2011).

- **shape:** Der übergebene Faktor bestimmt die Art der Datenpunktssymbole (vgl. Abschn. 14.3.1).
- **group:** Ein genannter Faktor sorgt dafür, dass Datenpunkte in hinzugefügten Schichten nach Faktorstufe getrennt behandelt werden, was etwa für Liniendiagramme relevant ist.

`ggplot()` repräsentiert ein Diagramm als Liste, die alle notwendigen Informationen beinhaltet und sich als Objekt speichern oder verändern lässt. Dagegen verwenden Diagrammfunktionen des Basisumfangs ein device wie eine Leinwand, auf der sie Elemente einzeichnen, erstellen aber keine abstrakte Repräsentation des Diagramms in Form eines Objekts. `ggplot`-Objekte werden erst dann als Diagramm in einem device dargestellt, wenn `print(ggplot-Objekt)` aufgerufen wird. Sie lassen sich besonders einfach mit `ggsave()` in Dateien unterschiedlichen Formats (etwa PDF, JPEG, PNG) speichern. Die hier vorgestellten Beispiele verwenden die folgenden simulierten Daten.

```
> Njk    <- 50                                # Gruppengröße
> P      <- 3                                 # Anzahl Stufen Gruppe
> Q      <- 2                                 # Anzahl Stufen Geschlecht
> IQ     <- rnorm(P*Q*Njk, mean=100, sd=15)   # IQ-Werte
> height <- rnorm(P*Q*Njk, mean=175, sd=7)    # Körpergröße
> rating <- sample(LETTERS[1:3], Njk*P*Q, replace=TRUE) # Rating

# Gruppenzugehörigkeit
> group <- factor(rep(c("control", "placebo", "treatment"), each=Q*Njk))
> sex   <- factor(rep(c("f", "m"), times=P*Njk))        # Geschlecht
> myDf  <- data.frame(sex, group, IQ, height, rating)  # Datensatz
```

Das von `ggplot()` erzeugte Objekt bildet die Grundsicht eines Diagramms, dem mit + zusätzlich weitere Schichten hinzugefügt werden müssen, um ein konkretes Diagramm zu erhalten und Diagrammoptionen zu ändern. Fügt man die Schichten schrittweise zu jeweils neuen Objekten zusammen, lässt sich für jeden Schritt kontrollieren, ob das Diagramm wie gewünscht aufgebaut wird.

Als Grundsicht soll hier der IQ gegen die Körpergröße aufgetragen werden, wobei das Geschlecht die Farbe der Datenpunktssymbole und die Gruppenzugehörigkeit die Art der Datenpunktssymbole bestimmt.

```
> library(ggplot2)                            # für ggplot()
> pA1 <- ggplot(myDf, aes(x=height, y=IQ, color=sex, shape=group))
```

Diagrammtypen

Jede der folgenden Funktionen fügt einer mit `ggplot()` erstellten Grundsicht einen konkreten Diagrammtyp hinzu (für weitere vgl. Chang, 2013). Dafür besitzen die genannten Funktionen eigene Argumente, um Eigenschaften zu kontrollieren, die spezifisch für den jeweiligen Diagrammtyp sind (s. u. für Beispiele und die zugehörige Hilfe-Seite für Details).

- `geom_point()` für ein Streudiagramm

- `geom_line()` für ein Liniendiagramm
- `geom_bar()` für ein Säulendiagramm
- `geom_histogram()` für ein Histogramm
- `geom_boxplot()` für einen boxplot

Als Beispiel soll zunächst mit `geom_point()` ein Streudiagramm erzeugt werden (Abb. 14.29).

```
# füge Grundschicht Streudiagramm mit größeren Datenpunktsymbolen hinzu
> pA2 <- pA1 + geom_point(size=3)
> print(pA2) # prüfe Zwischenstand
```

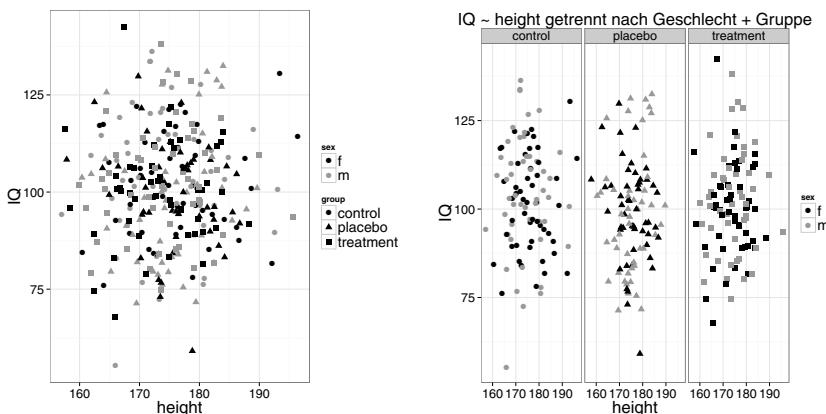


Abbildung 14.29: Streudiagramm mit `geom_point()`: Zwischenstand und nach Gruppen getrennte Endversion

Weitere Schichten dienen dazu, die Diagrammoptionen zu verändern.

- `ggtitle("<Text>")` für einen Diagrammtitel
- `xlab("<Text>")` und `ylab("<Text>")` für Beschriftungen der *x*- bzw. *y*-Achse
- `xlim(<min>, <max>)` und `ylim(<min>, <max>)` für Grenzen der *x*- bzw. *y*-Achse
- `theme(legend.position="none")` sorgt dafür, dass das Diagramm keine Legende enthält. Um die Legende an verschiedenen Stellen zu positionieren, kann `legend.position` auf "left", "right", "bottom" oder "top", gesetzt werden. Allgemein lassen sich mit `theme()` sehr viele Diagrammoptionen verändern.
- `guides(<Eigenschaft>=FALSE)` entfernt aus der Legende das zu `<Eigenschaft>` gehörende Element. So sorgt etwa `shape=FALSE` dafür, dass die Legende keine Einträge zur Art der verwendeten Datenpunktsymbolen enthält.

- Um die Diagrammregion in einzelne Facetten aufzuteilen, die jeweils einer Gruppe zugeordnet sind, kann eine mit `facet_grid(<Zeilen> ~ <Spalten>)` erstellte Schicht hinzugefügt werden. Dabei definiert die übergebene Modellformel die Aufteilung der Diagrammfläche in Zeilen und Spalten durch links oder rechts der `~` genannte Faktoren. Sollen Zeilen oder Spalten nicht unterteilt werden, ist statt eines Faktors ein `.` zu nennen.
- `facet_wrap(~ <Faktor>)` erstellt ebenfalls eine Facette für jede Gruppe des rechts der `~` genannten Faktors, ordnet die Facetten aber selbstständig in Form eines Rechtecks an.

Hier soll für jede Gruppe ein separates Streudiagramm erstellt und in Spalten der Diagrammfläche angeordnet werden. Da hier damit pro Facette nur eine Datenpunktssymbol-Art auftaucht, ist ein Legendeneintrag für diese Eigenschaft unnötig (Abb. 14.29).

```
> pA3 <- pA2 + facet_grid(. ~ group) # teile Diagrammfläche nach Gruppen
> pA4 <- pA3 + ggtitle("IQ ~ height getrennt nach Geschlecht + Gruppe")
> pA5 <- pA4 + guides(shape=FALSE)    # entferne Datenpunkt-Legende
> print(pA5)                         # fertiges Diagramm
```

Für ein Säulendiagramm mit `geom_bar()` ist im Aufruf von `ggplot(..., aes(x=(Variable)))` für `x` nur die Variable anzugeben, die die Höhe der Säulen bestimmt. In der üblichen Verwendung ist dies eine kategoriale Variable, deren Häufigkeiten von `geom_bar(stat="bin")` automatisch ausgezählt und als Säulenhöhe visualisiert werden. Dagegen stellt `geom_bar(stat="identity")` die nicht weiter aggregierten Werte von `x` selbst als Säulenhöhe dar – etwa wenn dies bereits Mittelwerte sind. `geom_bar(position=position_dodge())` sorgt für ein gruppiertes Säulendiagramm, während die Voreinstellung mit `position_stack()` ein gestapeltes ist (Abb. 14.30).

```
> pB1 <- ggplot(myDf, aes(x=rating, group=sex, fill=sex))
> pB2 <- pB1 + geom_bar(stat="bin", position=position_dodge())
> pB3 <- pB2 + facet_grid(. ~ group) # teile Diagrammfläche nach Gruppen
> pB4 <- pB3 + ggtitle("Rating-Häufigkeiten nach Geschlecht + Gruppe")
> print(pB4)                         # fertiges Diagramm
```

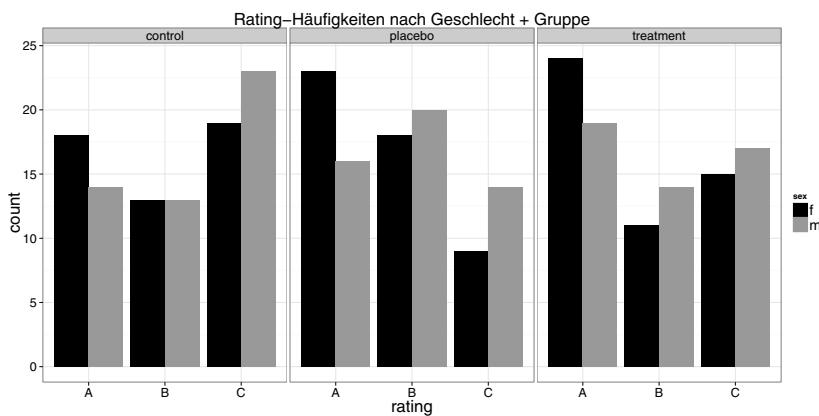


Abbildung 14.30: Nach Gruppen getrennte gruppierte Säulendiagramme mit `geom_bar()`

Bei einem Histogramm mit `geom_histogram()` ist ebenfalls im vorherigen Aufruf `ggplot(..., aes(x=<Variable>))` für `x` nur die Variable anzugeben, deren Häufigkeit von Werte-Intervallen das Histogramm darstellt. Über `geom_histogram(binwidth=<Breite>)` lässt sich die Breite der Intervalle kontrollieren (Abb. 14.31).

```
> pC1 <- ggplot(myDf, aes(x=IQ, fill=group))
> pC2 <- pC1 + geom_histogram()
> pC3 <- pC2 + facet_grid(. ~ group) # teile Diagrammfläche nach Gruppen
> pC4 <- pC3 + ggtitle("Histogramm IQ nach Gruppe")
> pC5 <- pC4 + theme(legend.position="none") # entferne Legende
> print(pC5) # fertiges Diagramm
```

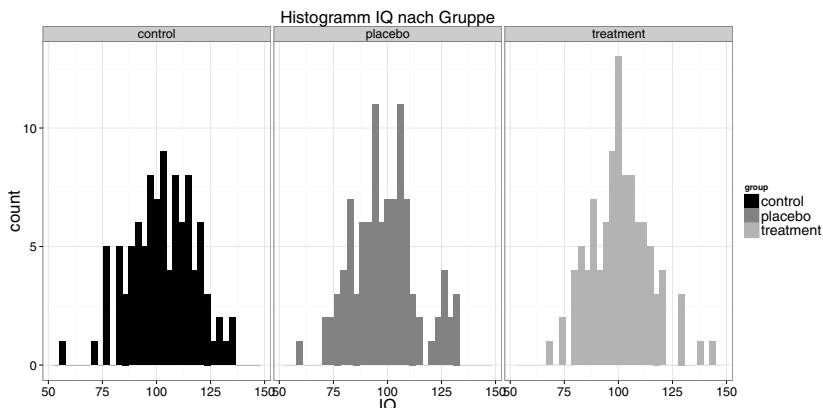


Abbildung 14.31: Nach Gruppen getrennte Histogramme mit `geom_histogram()`

`geom_boxplot()` erzeugt einen Boxplot, wobei im Aufruf `ggplot(..., aes(x=<Faktor>, y=<Variable>, ...))` für `x` ein Faktor zu übergeben ist, für dessen Stufen separate Boxplots angezeigt werden. Diese Boxplots stellen die Kennwerte der für `y` genannten Variable in den Gruppen dar (Abb. 14.32).

```
> pD1 <- ggplot(myDf, aes(x=sex, y=height, fill=sex))
> pD2 <- pD1 + geom_boxplot()
> pD3 <- pD2 + facet_grid(. ~ group) # teile Diagrammfläche nach Gruppen
> pD4 <- pD3 + ggtitle("Körpergröße nach Geschlecht + Gruppe")
> pD5 <- pD4 + theme(legend.position="none") # entferne Legende
> print(pD5) # fertiges Diagramm
```

Diagrammelemente hinzufügen

Einer Grundschicht mit konkretem Diagramm können weitere Schichten mit Diagrammelementen hinzugefügt werden, etwa vertikale bzw. horizontale Linien oder Regressionsgeraden (Abb. 14.33).

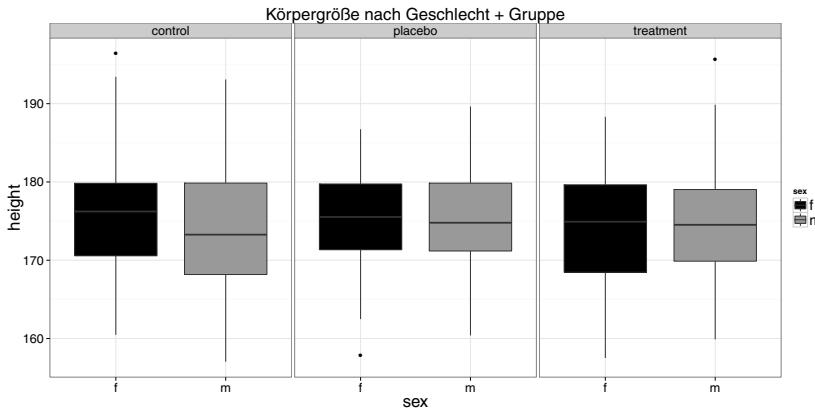


Abbildung 14.32: Nach Gruppen getrennte boxplots mit `geom_boxplot()`

- `geom_vline(aes(xintercept=(x-Koord.)))` für vertikale Linien, die die x -Achse in `xintercept` schneiden.
- `geom_hline(aes(yintercept=(y-Koord.)))` für horizontale Linien, die die y -Achse in `yintercept` schneiden.
- `geom_smooth(method=(Methode), se=TRUE, level=0.95, fullrange=FALSE)` für eine automatisch berechnete Regressionslinie. Dies ist etwa die Vorhersage einer linearen Regression mit `method=lm` (vgl. Abschn. 6.2) oder ein lokaler LOESS-Glätter mit `method=loess` (vgl. Abschn. 14.7.1, Voreinstellung für kleinere Datensätze). Mit `se=TRUE` erhält man zusätzlich den Konfidenzbereich für die Regressionslinie zum Niveau `level`. `fullrange=TRUE` sorgt dafür, dass sich die Regressionslinie bis an den Rand eines Diagramms über den beobachteten Wertebereich der x -Variable hinaus erstreckt.
- `geom_text(aes(x=(x-Koord.), y=(y-Koord.), label=(Variable)))` für Text, der in jeder Facette des Diagramms an den Koordinaten x und y erscheinen soll. Für `label` ist die Variable des Datensatzes im Aufruf von `ggplot()` zu nennen, die diesen Text für jede Beobachtung enthält.
- `annotate("text", x=(x-Koord.), y=(y-Koord.), label="Anmerkung")` für Textanmerkungen, die in jeder Facette des Diagramms identisch sind und an den Koordinaten x und y erscheinen sollen, wobei dieser Text an das Argument `label` zu übergeben ist.

```
> pE01 <- ggplot(myDf, aes(x=height, y=IQ, colour=sex:group, shape=sex))
> pE02 <- pE01 + geom_hline(aes(yintercept=100), linetype=2)
> pE03 <- pE02 + geom_vline(aes(xintercept=180), linetype=2)
> pE04 <- pE03 + geom_point(size=3)           # Streudiagramm
> pE05 <- pE04 + geom_smooth(method=lm, se=TRUE, size=1.2, fullrange=TRUE)
> pE06 <- pE05 + facet_grid(sex ~ group)      # teile in Facetten
> pE07 <- pE06 + ggtitle("IQ ~ height getrennt nach Geschlecht + Gruppe")
> pE08 <- pE07 + theme(legend.position="none")  # entferne Legende
```

```
# Anmerkung aus Variable im Datensatz
> pE09 <- pE08 + geom_text(aes(x=190, y=70, label=sgComb))

# Anmerkung direkt als Argument übergeben
> pE10 <- pE09 + annotate("text", x=165, y=130, label="Annotation")
> print(pE10) # fertiges Diagramm
```

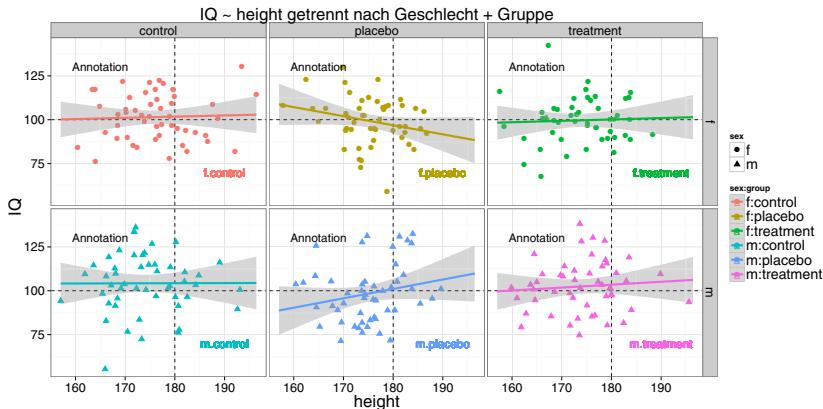


Abbildung 14.33: Nach Gruppen getrennte Streudiagramme inkl. Regressionsgerade und Konfidenzbereich mit `geom_point()`

14.8.4 Bedingte Diagramme für mehrere Gruppen mit lattice

Das Paket `lattice` stellt viele Funktionen bereit, um Diagramme zu erzeugen, die Daten getrennt nach Gruppen simultan in mehreren Facetten darstellen (Abb. 14.34, vgl. Abschn. 14.8.3, Fußnote 31). Diese Darstellungsart erlaubt es, alle hier bereits dargestellten Diagrammtypen zu verwenden, etwa Streudiagramme mit `xypplot()`, Säulendiagramme mit `barchart()`, dotcharts mit `dotplot()`, boxplots mit `bwplot()`, stripcharts mit `stripplot()` oder Histogramme mit `histogram()`. Der Aufruf all dieser Funktionen erfolgt immer in folgender Grundform:

```
> xyplot(<y-Koordinaten> ~ <x-Koordinaten> | <Faktor>, data=<Datensatz>)
```

Als erstes Argument ist eine Modellformel anzugeben, die wie in `plot()` zunächst die (x, y) -Koordinaten der darzustellenden Datenpunkte nennt. Als Besonderheit folgt dann hinter dem `|` Zeichen ein Faktor, der die Gruppen definiert, für die separate Diagramme erstellt werden sollen. Die Unterteilung kann auch durch die Kombination mehrerer Faktoren definiert sein, die dann in der Form `<Faktor1> * <Faktor2>` einzufügen sind. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Weitere Details enthalten die zugehörigen Hilfe-Seiten sowie Murrell (2011) und Sarkar (2008).

```
> library(lattice) # für histogram()
> histogram(IQ ~ height | sex*group, data=myDf,
+             main="Histogramme pro Gruppe")
```

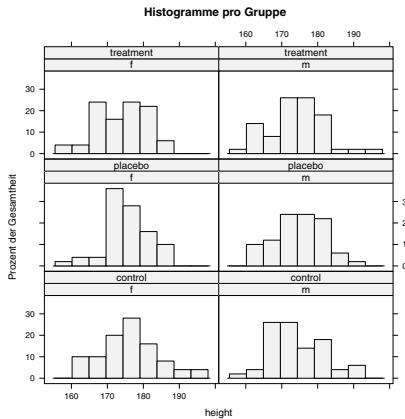


Abbildung 14.34: Darstellung multivariater Daten mit `histogram()`

14.8.5 Matrix aus Streudiagrammen

Enthält ein Datensatz viele Variablen, können ihre paarweise gebildeten gemeinsamen Verteilungen in einer Matrix aus Streudiagrammen simultan veranschaulicht werden, wie sie `pairs()` erzeugt. Die Matrix enthält jeweils so viele Zeilen und Spalten, wie Variablen vorhanden sind, wobei sich in einer Zelle (i, j) das Streudiagramm der gemeinsamen Verteilung der i -ten und j -ten Variable befindet. Auf der Hauptdiagonale (Zellen (i, i)) werden in der Voreinstellung die Variablennamen ausgegeben. An der Hauptdiagonale gespiegelte Zellen $((i, j)$ und (j, i)) stellen dieselbe gemeinsame Verteilung dar, allerdings mit vertauschten Achsen (Abb. 14.35).

```
> pairs(x=<Matrix>, labels=<Variablennamen>, panel=<Zeichenfunktion>,
+        lower.panel=<Zeichenfunktion>, upper.panel=<Zeichenfunktion>,
+        diag.panel=<Zeichenfunktion>)
```

Für `x` ist eine spaltenweise aus Variablen gebildete Matrix (oder ein Datensatz) anzugeben, deren paarweise gemeinsame Verteilungen dargestellt werden sollen. Ergeben sich die Variablennamen nicht aus den Spaltennamen von `x`, können sie als Vektor für `labels` genannt werden. `panel` erwartet den Namen einer Zeichenfunktion, deren Ergebnis in den Zellen außerhalb der Diagonale der erzeugten Grafik-Matrix abgebildet wird. Die Funktion muss als Argumente zwei Vektoren (Spalten von `x`) verarbeiten können und darf kein neues Diagramm öffnen, muss also eine Low-Level-Funktion sein. Voreinstellung ist `points` für ein Streudiagramm. Sollen die Zellen oberhalb und unterhalb der Hauptdiagonale der Grafik-Matrix unterschiedliche Diagrammtypen zeigen, können für `lower.panel` und `upper.panel` separate Zeichenfunktionen angegeben werden, die denselben Bedingungen wie jene für `panel` unterliegen. Durch Angabe einer Zeichenfunktion für `diag.panel`, die als Argument die Daten der Variable i erhält, lassen sich auch auf der Diagonale Diagramme einfügen (Voreinstellung ist `NULL` für die Ausgabe der Variablennamen).

Im Beispiel seien Personen in zwei Gruppen aufgeteilt und an ihnen vier Variablen erhoben worden. Die Streudiagramme sollen die Gruppenzugehörigkeit über die Farbe der Datenpunkt-symbole kenntlich machen (vgl. Abschn. 14.6.8).

```

> N <- 20                                     # Gesamt-N
> P <- 2                                      # Anzahl Gruppen
> IV <- rep(c("CG", "T"), each=N/P)          # Gruppenzugehörigkeit

# abhängige Variablen
> age    <- sample(18:35, N, replace=TRUE)
> IQ     <- round(rnorm(N, rep(c(100, 115), each=N/P), 15))
> rating <- round(0.4*IQ - 30 + rnorm(N, 0, 10), 1)
> score  <- round(-0.3*IQ + 0.7*age + rnorm(N, 0, 8), 1)
> mvDf   <- data.frame(IV, age, IQ, rating, score)      # Datensatz

# Matrix aus Streudiagrammen darstellen
> pairs(mvDf[c("age", "IQ", "rating", "score")],
+        main="Streudiagramm-Matrix", pch=16,
+        col=c("red", "blue")[unclass(mvDf$IV)])

```

Da sich nur Low-Level Zeichenfunktionen für die `panel` Argumente eignen, ist häufig der Umweg über selbst erstellte Funktionen notwendig, um bestimmte Grafiken in der Matrix darzustellen (vgl. Abschn. 15.2): Für High-Level-Funktionen muss dafür zunächst `par(new=TRUE)` aufgerufen werden, damit sie selbst kein neues device öffnen (vgl. Abschn. 14.9.2).

Hier soll dies zunächst für in der Hauptdiagonale darzustellende Histogramme geschehen. Zusätzlich sollen die für beide Gruppen getrennten Streuungsellipsen der gemeinsamen Verteilung in den Zellen oberhalb der Hauptdiagonale gezeigt werden (Abb. 14.24, vgl. Abschn. 14.6.8).

```

# Funktion, um Histogramm darzustellen, ohne neue Grafik zu öffnen
> myHist <- function(x, ...) { par(new=TRUE); hist(x, ..., main="") }

# Funktion, um Streuungsellipsen getrennt für beide Gruppen zu zeichnen
> myEll <- function(x, y, nSegments=100, rad=1, ...) {
+   # trenne Beobachtungen in Variablen x und y nach Gruppen und
+   # verbinde Variablen getrennt nach Gruppen zu einer Datenmatrix
+   splLL <- split(data.frame(x, y), mvDf$IV)
+   matCG <- data.matrix(splLL$CG)           # Daten Gruppe 1
+   matT  <- data.matrix(splLL$T)            # Daten Gruppe 2
+   ctrCG <- colMeans(matCG)                # Zentroid Gruppe 1
+   ctrT  <- colMeans(matT)                 # Zentroid Gruppe 2
+
+   # für jede Gruppe: Cholesky-Dekomposition der Kovarianzmatrix
+   RRcg <- chol(cov(matCG))
+   RRt  <- chol(cov(matT))
+
+   # Ellipse: zunächst Winkel im Einheitskreis
+   angles <- seq(0, 2*pi, length.out=nSegments)
+
+   # rotiere Ellipse im Ursprung, vergrößere um Faktor rad
+   ellCG <- rad * cbind(cos(angles), sin(angles)) %*% RRcg
+   ellT  <- rad * cbind(cos(angles), sin(angles)) %*% RRt

```

```

+
# verschiebe Ellipse ins Zentroid der jeweiligen Daten
+ ellCGctr <- sweep(ellCG, 2, ctrCG, "+")
+ ellTctr <- sweep(ellT, 2, ctrT, "+")
+
# zeichne Ellipse und Zentroid für jede Gruppe
+ points(ctrCG[1], ctrCG[2], col="red", pch=4, lwd=2)
+ points(ctrT[1], ctrT[2], col="blue", pch=4, lwd=2)
+ polygon(ellCGctr, border="red")      # Ellipse Gruppe 1
+ polygon(ellTctr, border="blue")       # Ellipse Gruppe 2
}

> pairs(mvDf[,c("age", "IQ", "rating", "score")], diag.panel=myHist,
+        upper.panel=myEll, main="Streudiagramm-Matrix", pch=16,
+        col=c("red", "blue"))[unclass(mvDf$IV)])

```

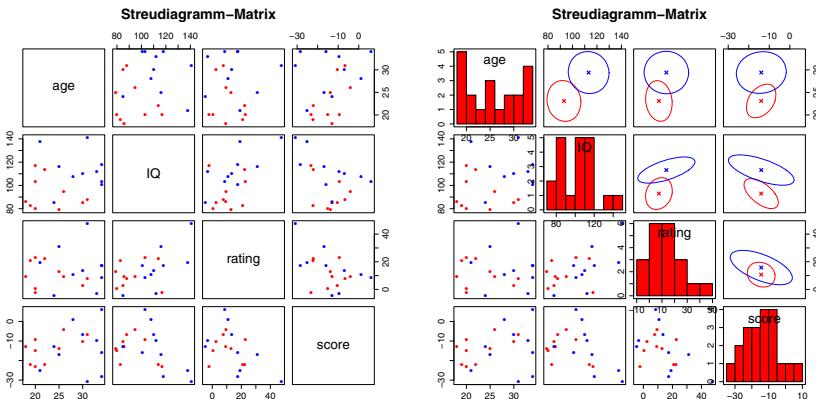


Abbildung 14.35: Matrizen von paarweisen Streudiagrammen mehrerer Variablen mit `pairs()`

14.8.6 Heatmap

Eine *heatmap* ist eine Falschfarben-Darstellung der Werte einer Matrix, wobei für jede Zelle eine Farbe entsprechend der Größe des Zellenwertes gewählt wird. Mit `heatmap()` können auf diese Weise etwa die Einträge der Korrelationsmatrix vieler Variablen so visualisiert werden, dass unmittelbar ersichtlich ist, welche Variablen einen starken linearen Zusammenhang aufweisen.

```

> heatmap(<Matrix>, symm, revC, Rowv, Colv, recV, col,
+         RowSideColors, ColSideColors)

```

Das erste Argument erwartet eine numerische Matrix. Handelt es sich um eine symmetrische Matrix (z. B. eine Korrelationsmatrix), kann dies mit `symm=TRUE` angezeigt werden. In diesem Fall sorgt `revC=TRUE` dafür, dass die Reihenfolge von Zeilen und Spalten übereinstimmt. `Rowv`

und `Colv` kontrollieren, ob an den Grafikrändern ein *Dendrogramm* angezeigt werden soll, das die mit einer Clusteranalyse ermittelten hierarchischen Zusammenhänge von Variablen darstellt. Setzt man diese Argumente auf `NA`, wird kein Dendrogramm angezeigt. Die zu verwendenden Farben können an `col` übergeben werden, wobei soviele Farben wie Zellen vorhanden sein müssen. Um die Reihenfolge der Farben in einer Legende an den Seiten des Diagramms darzustellen, können sie an `RowSideColors` (soviele Farben wie Zeilen) oder `ColSideColors` (soviele Farben wie Spalten) übergeben werden.

Die Ausgabe entspricht der Struktur der übergebenen Matrix, wobei jede Zelle durch eine Farbe dargestellt wird (Abb. 14.36). Farben werden Werten so zugeordnet, dass die erste Farbe in `col` zum kleinsten Wert gehört und die letzte Farbe zum größten Wert in der Matrix.

```
> corMat <- cor(data.matrix(mvDf))          # Korrelationsmatrix
> round(corMat, digits=3)
      IV   age    IQ rating score
IV  1.000 0.578 0.570 0.209 0.003
age 0.578 1.000 0.210 0.041 0.254
IQ   0.570 0.210 1.000 0.546 -0.435
rating 0.209 0.041 0.546 1.000 -0.458
score  0.003 0.254 -0.435 -0.458 1.000

> cmCol <- cm.colors(nrow(corMat))          # Basisfarben
> cmRamp <- colorRampPalette(cmCol)          # Funktion für Mischfarben
> cols <- cmRamp(length(corMat))             # Mischfarben
> heatmap(corMat, symm=TRUE, revC=TRUE, Rowv=NA, Colv=NA, col=cols,
+           RowSideColors=cmCol, main="Heatmap einer Korrelationsmatrix")
```

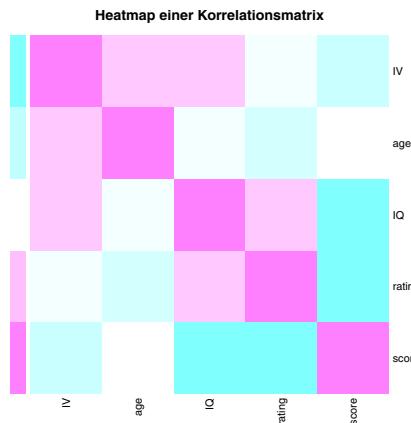


Abbildung 14.36: Heatmap einer Korrelationsmatrix

`heatmap()` verfügt über viele weitere Darstellungsmöglichkeiten, die insbesondere im Rahmen einer Clusteranalyse sinnvoll sind und in `?heatmap` erläutert werden.

14.9 Mehrere Diagramme in einem Grafik-Device darstellen

Die in Abschn. 14.8.3, 14.8.4 und 14.8.5 vorgestellten Funktionen verdeutlichen, dass auch mehr als ein Diagramm in ein device gezeichnet werden kann. Sollen nicht nur Diagramme gleichen Typs, sondern auch unterschiedliche Grafiken in einem device dargestellt werden, lässt sich dessen Fläche manuell in mehrere rechteckige Bereiche aufteilen. Diese können dann mit jeweils einem Diagramm gefüllt werden. Auf diese Weise simultan dargestellte Diagramme erleichtern den Vergleich von Ergebnissen in verschiedenen Teilstichproben, lassen sich aber auch nutzen, um dieselben Daten parallel auf verschiedene Arten zu visualisieren.

14.9.1 layout()

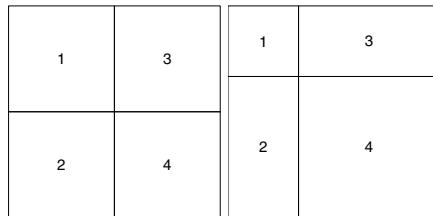


Abbildung 14.37: Mit `layout()` eingeteilte Regionen eines device

Um n Diagramme in einem device darzustellen, kann dessen Fläche mit `layout()` unter Zuhilfenahme einer Matrix `mat` aufgeteilt werden, die aus den Zahlen $0, 1, \dots, n$ gebildet ist. Die Ergebnisse der sich an `layout()` anschließenden n High-Level-Funktionen zum Erstellen von Diagrammen werden automatisch den durch `mat` definierten Regionen zugewiesen, statt jeweils die gesamte Fläche des aktiven device zu überschreiben (Abb. 14.37, 14.38).

```
> layout(mat=<Matrix>, widths=<Vektor>, heights=<Vektor>)
```

`mat` bestimmt die Einteilung der Device-Region in Planquadrate, die durch die Zellen von `mat` symbolisiert werden. Enthält eine Zelle von `mat` dabei die 0, wird die zugehörige Region beim Befüllen mit Grafiken übersprungen und bleibt leer. Dann ist die Anzahl n der dargestellten Diagramme kleiner als die Zahl der Zellen von `mat`, also der ursprünglichen Planquadrate. Über das Argument `widths` ist die Breite der Spalten und über `heights` die Höhe der Zeilen veränderbar – in der Voreinstellung erfolgt die Aufteilung gleichmäßig (Abb. 14.39). Breite und Höhe können zum einen als relative Größe in Form eines Vektors von ganzzahligen Gewichten ausgedrückt werden, die dann an der Summe aller Gewichte relativiert werden. Zum anderen können die Maße auch als absolute Werte mit `1cm(<Zahl>)` in der Einheit cm eingegeben werden. Um die n Planquadrate durch Umrandung sichtbar zu machen, ist `layout.show(n)` zu verwenden.

```
> (mat1 <- matrix(1:4, 2, 2))    # vier Regionen derselben Größe definieren
 [,1] [,2]
[1,] 1     3
[2,] 2     4
```

```

> layout(mat1)                                # Device-Fläche teilen
> layout.show(4)                             # Regionen anzeigen

# dieselben Regionen mit unterschiedlicher Größe
> layout(mat1, widths=c(1, 2), heights=c(1, 2))    # Aufteilung 1/3, 2/3
> layout.show(4)                             # Regionen anzeigen

# vier Diagramme einfügen
> barplot(table(round(rnorm(100))), horiz=TRUE, main="Balkendiagramm")
> boxplot(rt(100, 5), main="Boxplot")
> stripchart(sample(1:20, 40, replace=TRUE), method="stack",
+             main="Stripchart")

> pie(table(sample(1:6, 20, replace=TRUE)), main="Kreisdiagramm")

```

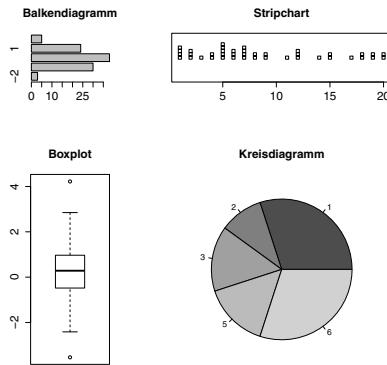


Abbildung 14.38: Verschiedene Diagramme in einem Diagrammfenster mittels `layout()`

Benachbarte Planquadrate der Device-Fläche lassen sich zusammenfassen, indem den zugehörigen Matrixelementen dieselbe Zahl zugeordnet wird (Abb. 14.39). Auch hier ist die Anzahl n der dargestellten Diagramme kleiner als die Zahl der Zellen von `mat`.

```

# Zellen der ersten Zeile zusammenfassen, Region links unten leer
> (mat2 <- matrix(c(1, 0, 1, 2), 2, 2))
      [,1] [,2]
[1,]    1    1
[2,]    0    2

> layout(mat2)
> stripchart(sample(1:20, 40, replace=TRUE), method="stack",
+             main="Stripchart")

> barplot(table(round(rnorm(100))), main="Säulendiagramm")

```

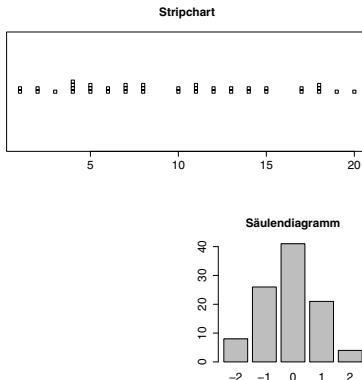


Abbildung 14.39: Mehrere, teils zusammengefasste, teils leere Regionen in einem Diagrammfenster mittels `layout()`

14.9.2 `par(mfrow, mfcoll, fig)`

Ein weiterer Weg Device-Flächen zu unterteilen, besteht in der Verwendung der Argumente `mfrow` oder `mfcoll` von `par()`.

```
> par(mfrow=c(<Anzahl Zeilen>, <Anzahl Spalten>))
> par(mfcoll=c(<Anzahl Zeilen>, <Anzahl Spalten>))
```

An `mfrow` oder `mfcoll` ist ein Vektor mit zwei Elementen zu übergeben, die die Anzahl der Zeilen und Spalten definieren, aus denen sich dann die einzelnen Regionen der Device-Fläche ergeben. Diese Regionen besitzen dieselbe Größe und lassen sich nicht weiter zusammenfassen. Der Unterschied zwischen beiden Argumenten betrifft die Reihenfolge, in der die entstehenden Regionen mit Diagrammen gefüllt werden: Mit `mfrow` werden durch sich anschließende High-Level-Funktionen nacheinander zunächst die Zeilen, bei `mfcoll` entsprechend nacheinander die Spalten gefüllt. Sind auf der Device-Fläche in einem rechteckigen Layout letztlich `<Anzahl>` viele Diagramme unterzubringen, kann alternativ auch die Funktion `n2mfrow(<Anzahl>)` zur Erstellung eines geeigneten Vektors verwendet werden, der dann an `mfrow` zu übergeben ist.

```
> dev.new(width=7, height=4)                      # Diagrammgröße ändern
> par(mfrow=c(1, 2))                            # Diagrammfenster aufteilen
> boxplot(rt(100, 5), xlab=NA, notch=TRUE, main="Boxplot")
> plot(rnorm(10), pch=16, xlab=NA, ylab=NA, main="Streudiagramm")
```

Mit `par(fig=<Vektor>, new=TRUE)` kann alternativ vor einem High-Level-Befehl zum Erstellen eines Diagramms die rechteckige Figure-Region innerhalb der Device-Fläche definiert werden, in die das folgende Diagramm gezeichnet wird.

Die Elemente des Vektors `fig` stellen dabei die Koordinaten der Rechteckkanten in der Reihenfolge links, rechts, unten, oben dar und müssen Werte im Bereich von 0–1 besitzen. Der für die untere und obere Kante eingetragene Wert gilt als deren jeweilige *y*-Koordinate, wobei

0 die Koordinate des unteren und 1 die des oberen Device-Randes ist. Der für die linke und rechte Kante eingetragene Wert gilt als deren jeweilige x -Koordinate, wobei 0 die Koordinate des linken und 1 die des rechten Device-Randes ist. Das Argument `new=TRUE` legt fest, dass der folgende Grafikbefehl dem aktiven device hinzugefügt werden soll. Indem `par(fig)` mehrfach alternierend mit High-Level-Grafikfunktionen aufgerufen wird, lassen sich mehrere Diagramme in einem device plazieren (Abb. 14.40).

```
# Simulation von 1000 mal 10 Bernoulli-Experimenten
> resBinom <- rbinom(1000, size=10, prob=0.3)

# Ergebnis in Faktor umwandeln, um Kategorien hinzufügen zu können,
# die möglich sind, aber nicht auftauchen
> facBinom <- factor(resBinom, levels=0:10)

# Häufigkeitsverteilung bestimmen und als senkrechte Linien darstellen
> tabBinom <- table(facBinom)
> par(fig=c(0, 1, 0.10, 1))
> plot(tabBinom, type="h", bty="n", xlim=c(0, 10), xlab=NA,
+      xaxt="n", ylab="Häufigkeit", main="Ergebnisse von
+      1000*10 Bernoulli Experimenten (p=0.3)")

# Werte zusätzlich als Punkte einzeichnen
> points(names(tabBinom), tabBinom, pch=16, cex=2, col="gray")

# im unteren Diagrammbereich gekerbten Boxplot einzeichnen
> par(fig=c(0, 1, 0, 0.35), bty="n", new=TRUE)
> boxplot(resBinom, horizontal=TRUE, ylim=c(0, 10), notch=TRUE,
+          col="gray", xlab="Anzahl der Erfolge")
```

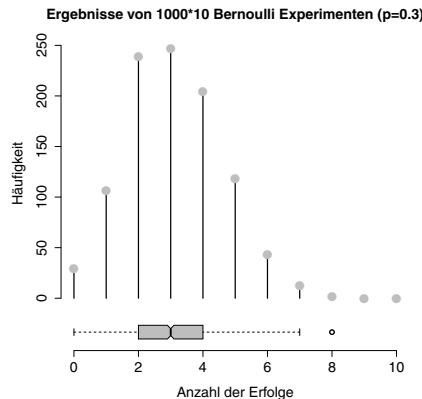


Abbildung 14.40: Anpassen der Größe verschiedener Regionen mit `par(fig)`

14.9.3 `split.screen()`

Eine dritte Methode, um die Device-Fläche in Regionen zu unterteilen und mit separaten Diagrammen zu füllen, stellt die `split.screen()` Funktion bereit, die in Kombination mit `screen()` und `close.screen()` zu verwenden ist.

```
> split.screen(figs=<2-Vektor oder Nx4-Matrix>, screen=<Nummer>)
```

Das Argument `figs` erwartet einen Vektor mit zwei Elementen oder eine Matrix mit vier Spalten, die beide die Aufteilung der Device-Fläche definieren können. Ein Vektor legt durch seine beiden Elemente die Zahl der Zeilen und Spalten fest, wie dies auch in `par(mfrow)` geschieht. Die Fläche wird in diesem Fall gleichmäßig unter den Regionen aufgeteilt. Dagegen wird jede der in einer für `figs` übergebenen vierstöckigen Matrix enthaltenen Zeilen als Definition jeweils einer rechteckigen Region interpretiert. Die Werte in einer Zeile stellen dabei die Koordinaten der Rechteckkanten in der Reihenfolge links, rechts, unten, oben dar und müssen im Bereich von 0–1 liegen. Der für die untere und obere Kante eingetragene Wert gilt als deren jeweilige y -Koordinate, wobei 0 die Koordinate des unteren und 1 die des oberen Device-Randes ist. Der für die linke und rechte Kante eingetragene Wert gilt als deren jeweilige x -Koordinate, wobei 0 die Koordinate des linken und 1 die des rechten Device-Randes ist. Auf diese Weise können auch sich überschneidende, oder die Diagrammfläche nicht vollständig ausfüllende Regionen definiert werden (Abb. 14.41).

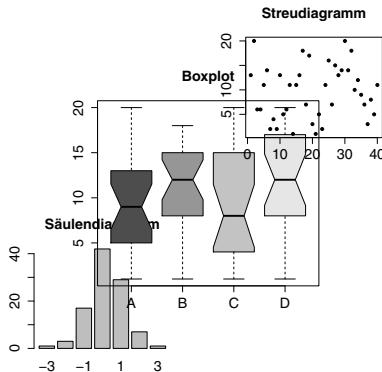


Abbildung 14.41: Sich überlappende Regionen mit `screen()` erzeugen

Die mit `split.screen()` gebildeten Regionen sind intern bei 1 beginnend nummeriert und können über ihre Nummer angesprochen werden. Dies ist in zwei Fällen notwendig: Zum einen kann eine Teilfläche durch erneuten Aufruf von `split.screen(screen=<Nummer>)` weiter unterteilt werden, wenn dabei an das Argument `screen` die Nummer der zu unterteilenden Fläche übergeben wird. Zum anderen muss mit `screen(<Nummer>)` vor jedem folgenden Befehl zur Diagrammerstellung die Nummer der Fläche genannt werden, in die das Diagramm gezeichnet werden soll.

Eine Teilfläche sollte vollständig bearbeitet werden, ehe die nächste Region ausgewählt wird. Zwar ist es prinzipiell möglich, eine bereits verlassene Teilfläche wieder zu aktivieren, aller-

dings mit dem Risiko unerwünschter Effekte. Der Inhalt einer mit `split.screen()` erzeugten Teilfläche kann mit `erase.screen(<Nummer>)` wieder gelöscht werden.

Um eine mit `split.screen()` erzeugte Teilfläche abzuschließen und damit ein weiteres Hinzufügen von Elementen zu verhindern, dient `close.screen(<Nummer>, all.screens=FALSE)`. Mit ihr können über das Argument `all.screens=TRUE` auch alle Teilebenen simultan gesperrt werden. Dies sollte etwa geschehen, wenn alle Regionen eines device mit Diagrammen gefüllt und keine weiteren Änderungen gewünscht sind.

```
# 3 Regionen als Zeilen einer Matrix definieren
> splitMat <- rbind(c(0.0, 0.5, 0.0, 0.5), c(0.15, 0.85, 0.15, 0.85),
+                      c(0.5, 1.0, 0.5, 1.0))

> split.screen(splitMat)                      # Diagrammfläche unterteilen
> screen(1)                                    # in Region 1 zeichnen
> barplot(table(round(rnorm(100))), main="Säulendiagramm")
> screen(2)                                    # in Region 2 zeichnen
> boxplot(sample(1:20, 100, replace=TRUE) ~ gl(4, 25, labels=LETTERS[1:4]),
+           col=rainbow(4), notch=TRUE, main="Boxplot")

> screen(3)                                    # in Region 3 zeichnen
> plot(sample(1:20, 40, replace=TRUE), pch=20, xlab=NA, ylab=NA,
+       main="Streudiagramm")

> close.screen(all.screens=TRUE)                # Bearbeitung abschließen
```

Kapitel 15

R als Programmiersprache

R bietet nicht nur Mittel zur numerischen und grafischen Datenanalyse, sondern ist gleichzeitig eine Programmiersprache, die dieselbe Syntax wie die bisher behandelten Auswertungen verwendet. Das seinerseits sehr umfangreiche Thema der Programmierung mit R soll in den folgenden Abschnitten nur soweit angedeutet werden, dass nützliche Sprachkonstrukte wie z. B. Kontrollstrukturen verwendet sowie einfache Funktionen selbst erstellt und analysiert werden können. Eine ausführliche Behandlung sei der hierauf spezialisierten Literatur überlassen (Chambers, 2008; Ligges, 2014; Wickham, 2014a). Die Entwicklung eigener R-Pakete behandeln R Development Core Team (2014d) und Wickham (2014c).

15.1 Kontrollstrukturen

Kontrollstrukturen ermöglichen es, die Abfolge von Befehlen gezielt zu steuern. Sie machen etwa die Ausführung von Befehlen von Bedingungen abhängig und verzweigen so den Befehlsfluss, oder wiederholen in *Schleifen* dieselben Befehle, solange bestimmte Nebenbedingungen gelten.¹

15.1.1 Fallunterscheidungen

Mit `if()` können Befehle im Rahmen einer Fallunterscheidung in Abhängigkeit davon ausgeführt werden, ob eine Bedingung gilt.

```
> if(<logischer Ausdruck>) {  
+   <Befehlsblock>           # ausgeführt, wenn <Ausdruck> TRUE ist  
+ }
```

Als Argument erwartet `if()` einen Ausdruck, der sich zu einem einzelnen Wahrheitswert `TRUE` oder `FALSE` auswerten lässt. Hierbei ist auszuschließen, dass der Ausdruck einen Vektor der Länge 0, `NA`, `NaN` oder `NULL` ergibt – etwa indem der Ausdruck in `isTRUE()` eingeschlossen wird (vgl. Abschn. 1.3.6).

Mögliche Ausdrücke sind etwa logische Vergleiche, wobei im Fall eines auf diese Weise erzeugten Vektors von Wahrheitswerten nur dessen erstes Element berücksichtigt wird. Im folgenden, durch `{}` eingeschlossenen Block, sind jene Befehle aufzuführen, die nur dann ausgewertet

¹Für Hilfe zu diesem Thema vgl. `?Control`.

werden sollen, wenn der Ausdruck gleich TRUE ist.² Finden die Befehle in einer Zeile hinter `if()` Platz, sind die geschweiften Klammern optional.³ Das Ergebnis von `if()` ist der Wert des letzten ausgewerteten Ausdrucks und lässt sich direkt einem Objekt zuweisen.

```
> (x <- round(rnorm(1, 100, 15)))
[1] 115
```

```
> y <- if(x > 100) { TRUE }
> y
[1] TRUE
```

Für den Fall, dass der Ausdruck gleich FALSE ist, kann eine Verzweigung des Befehlsflusses auch einen alternativen Block von Befehlen vorsehen, der sich durch das Schlüsselwort `else` eingeleitet an den auf `if()` folgenden Block anschließt.

```
> if(<Ausdruck>) {
+   <Befehlsblock>                      # ausgeführt, wenn <Ausdruck> TRUE ist
+ } else {
+   <Befehlsblock>                      # ausgeführt, wenn <Ausdruck> FALSE ist
+ }
```

Hier ist zu beachten, dass sich das Schlüsselwort `else` in derselben Zeile wie die schließende Klammer } des `if()` Blocks befinden muss und nicht separat in der auf die Klammer folgenden Zeile stehen kann.

```
> x <- round(rnorm(1, 100, 15))
> if(x > 100) {
+   cat("x is", x, "(greater than 100)\n")
+ } else {
+   cat("x is", x, "(100 or less)\n")
+ }
x is 83 (100 or less)
```

Innerhalb der auf `if()` oder `else` folgenden Befehlssequenz können wiederum `if()` Abfragen stehen. So verschachtelt lassen sich auch Bedingungen prüfen, die mehr als zwei Ausprägungen annehmen können.

```
> (day <- sample(c("Mon", "Tue", "Wed"), 1))
[1] "Wed"

> if(day == "Mon") {
+   print("The day is Monday")
+ } else {
```

²Mit `if(FALSE) { <Befehle> }` können damit schnell viele Befehlszeilen von der Verarbeitung ausgeschlossen werden, ohne diese einzeln mit `#` auskommentieren zu müssen. Die ausgeschlossenen Zeilen müssen dabei jedoch nach wie vor syntaktisch korrekt, können also keine Kommentare im engeren Sinne sein.

³Allgemein gilt, dass in einer Zeile stehende Befehle einen Block bilden und zusammen ausgeführt werden. Geschweifte Klammern sorgen dafür, dass die zwischen ihnen stehenden Befehle auch dann als einzelner Block gewertet werden, wenn sie sich über mehrere Zeile erstrecken. Die Auswertung erfolgt erst, wenn die Klammer geschlossen wird, auch wenn Abschnitte davor bereits für sich genommen syntaktisch vollständig sind.

```
+   if(day == "Tue") {
+     print("The day is Tuesday")
+   } else {
+     print("The day is neither Monday nor Tuesday")
+   }
+ }
[1] "The day is neither Monday nor Tuesday"
```

Die `switch()` Anweisung ist für eben solche Situationen gedacht, in denen eine Nebenbedingung mehr als zwei Ausprägungen besitzen kann und für jede dieser Ausprägungen anders verfahren werden soll. `switch()` ist meist übersichtlicher als eine ebenfalls immer mögliche verschachtelte `if()` Abfrage. Lässt sich die Bedingung zu einem ganzzahligen Wert im Bereich von 1 bis zur Anzahl der möglichen Befehle auswerten, hat die Syntax folgende Form:

```
> switch(EXPR=<Ausdruck>, <Befehl 1>, <Befehl 2>, ...)
```

Als erstes Argument `EXPR` ist der Ausdruck zu übergeben, von dessen Ausprägung abhängen soll, welcher Befehl ausgeführt wird – häufig ist dies lediglich ein Objekt. Es folgen durch Komma getrennt mögliche Befehle. `EXPR` muss in diesem Fall den auszuführenden Befehl numerisch bezeichnen, bei einer 1 würde der erste Befehl ausgewertet, bei einer 2 der zweite, usw. Sollen für einen Wert von `EXPR` mehrere Befehle ausgeführt werden, sind diese in {} einzuschließen.

```
> (val <- sample(1:3, 1))
[1] 3
```

```
> switch(val, print("val is 1"), print("val is 2"), print("val is 3"))
[1] "val is 3"
```

Ist `EXPR` dagegen eine Zeichenkette, hat die Syntax die folgende Gestalt.

```
> switch(EXPR=<Ausdruck>, <Wert 1>=<Befehl 1>, <Wert 2>=<Befehl 2>, ...
+       <Befehl>)
```

Auf `EXPR` folgt hier durch Komma getrennt eine Reihe von (nicht in Anführungszeichen stehenden) möglichen Werten mit einem durch Gleichheitszeichen angeschlossenen zugehörigen Befehl. Schließlich besteht die Möglichkeit, einen Befehl ohne vorher genannte Ausprägung für all jene Situationen anzugeben, in denen `EXPR` keine der zuvor explizit genannten Ausprägungen besitzt – andernfalls ist das Ergebnis in einem solchen Fall `NULL`.

```
> myCalc <- function(op, vals) {
+   switch(op,
+     plus = vals[1] + vals[2],
+     minus = vals[1] - vals[2],
+     times = vals[1] * vals[2],
+     vals[1] / vals[2])
+ }

> (vals <- round(rnorm(2, 1, 10)))
[1] -5 -7
```

```
> myCalc("minus", vals)
[1] 2
```

```
> myCalc("XX", vals)
[1] 0.7142857
```

15.1.2 Schleifen

Schleifen dienen dazu, eine Folge von Befehlen mehrfach ausführen zu lassen, ohne diese Befehle für jede Ausführung neu notieren zu müssen. Wie oft eine Befehlssequenz durchlaufen wird, kann dabei von Nebenbedingungen und damit von zuvor durchgeföhrten Auswertungen abhängig gemacht werden.⁴

```
> for(<Variable> in <Vektor>) {
+   <Befehlsblock>
+ }
```

Die Funktion `for()` besteht aus zwei Teilen: zum einen, in runde Klammern eingeschlossen, dem *Kopf* der Schleife, zum anderen, darauf folgend in `{}` eingeschlossen, der zu wiederholenden Befehlssequenz – dem *Rumpf*. Im Kopf der Schleife ist `<Variable>` ein Objekt, das im Rumpf verwendet werden kann. Es nimmt nacheinander die in `<Vektor>` enthaltenen Werte an, oft ist dies eine numerische Sequenz. Für jedes Element von `<Vektor>` wird der Schleifenrumpf einmal ausgeführt, wobei der Wert von `<Variable>` wie beschrieben nach jedem Durchlauf der Schleife wechselt.

```
> ABC <- c("Alfa", "Bravo", "Charlie", "Delta")
> for(i in ABC) { print(i) }
[1] "Alfa"
[1] "Bravo"
[1] "Charlie"
[1] "Delta"
```

Mit Schleifen lassen sich Simulationen erstellen, mit denen etwa die Robustheit statistischer Verfahren bei Verletzung ihrer Voraussetzungen untersucht werden kann. Im folgenden Beispiel wird zu diesem Zweck zunächst eine Grundgesamtheit von 100000 Zufallszahlen einer normalverteilten Variable simuliert. Daraufhin werden die Zahlen quadriert und logarithmiert, also einer nichtlinearen Transformation unterzogen. In einer Schleife werden aus dieser Grundgesamtheit 1000 mal 2 Gruppen von je 20 Zahlen zufällig ohne Zurücklegen gezogen und mit einem *t*-Test für unabhängige Stichproben sowie mit einem *F*-Test auf Varianzhomogenität verglichen. Da beide Stichproben aus derselben Grundgesamtheit stammen, ist in beiden Tests die Nullhypothese richtig, die Voraussetzung der Normalverteiltheit dagegen verletzt. Ein robuster Test sollte in dieser Situation nicht wesentlich häufiger fälschlicherweise signifikant werden, als durch das

⁴ Anders als in kompilierten Programmiersprachen wie etwa C oder Fortran sind Schleifen in R als interpretierter Sprache oft ineffizient. Als Grundregel sollten sie deswegen bei der Auswertung größerer Datenmengen nach Möglichkeit vermieden und durch *vektorierte* Befehle ersetzt werden, die mehrere, als Vektor zusammenfasste Argumente gleichzeitig bearbeiten (Ligges & Fox, 2008).

nominelle α -Niveau vorgegeben. Während der t -Test im Beispiel in der Tat robust ist, reagiert der F -Test sehr liberal – er fällt weit häufiger signifikant aus, als das nominelle α -Niveau erwarten lässt.

```
> src      <- log((rnorm(100000, 0, 1))^2)      # Grundgesamtheit
> alpha    <- 0.05                                # nominelles alpha
> nTests   <- 1000                                # Anzahl simulierter Tests
> Nj       <- 20                                  # Gruppengröße
> sigVecT <- numeric(nTests)                      # für Ergebnisse t.test()
> sigVecV <- numeric(nTests)                      # für Ergebnisse var.test()

> for(i in seq(length.out=nTests)) {
+   group1     <- sample(src, Nj, replace=FALSE)      # Stichprobe 1
+   group2     <- sample(src, Nj, replace=FALSE)      # Stichprobe 2
+   resT       <- t.test(group1, group2)              # t-Test
+   resV       <- var.test(group1, group2)             # Varianz-Test
+   sigVecT[i] <- as.logical(resT$p.value < alpha)  # t signifikant?
+   sigVecV[i] <- as.logical(resV$p.value < alpha)  # F signifikant?
+ }

> cat("Erwartete Anzahl signifikanter Tests:", alpha*nTests, "\n")
Erwartete Anzahl signifikanter Tests: 50

> cat("Signifikante t-Tests:", sum(sigVecT), "\n")
Signifikante t-Tests: 46

> cat("Signifikante F-Tests auf Varianzhomogenität:", sum(sigVecV), "\n")
Signifikante F-Tests auf Varianzhomogenität: 197
```

Im konkreten Beispiel ließe sich die `for()` Schleife auch vermeiden, indem stärker R-eigene Konzepte zur häufigen Wiederholung derselben Arbeitsschritte sowie Möglichkeiten zur Vektorisierung genutzt werden. Häufig ist insbesondere letzteres aus Effizienzgründen erstrebenswert. Schleifen können dann gut vermieden werden, wenn die Berechnungen in den einzelnen Schleifendurchläufen voneinander unabhängig sind, wenn – wie hier – ein Durchlauf also keine Werte benötigt, die in vorherigen Durchläufen berechnet wurden.

```
# wiederhole nTests mal für beide Stichproben das Ziehen von je Nj Personen
> group1 <- replicate(nTests, sample(src, Nj, replace=FALSE))
> group2 <- replicate(nTests, sample(src, Nj, replace=FALSE))
> groups <- rbind(group1, group2)      # füge Daten zu Matrix zusammen

# berechne Teststatistiken für t-Tests: zunächst Streuungsschätzungen
> estSigDiffs <- apply(groups, 2, function(x) {
+   sqrt(1/Nj) * sqrt(var(x[1:Nj]) + var(x[(Nj+1):(2*Nj)])) } )

# berechne alle Mittelwertsdifferenzen
> meanDiffs <- apply(groups, 2, function(x) {
+   mean(x[1:Nj]) - mean(x[(Nj+1):(2*Nj)]) } )
```

```
# vektorisierte Befehle, um t-Werte und p-Werte zu erhalten
> tVals <- meanDiff / estSigDiff      # t-Werte
> pVals <- pt(tVals, (2*Nj)-2, lower.tail=FALSE)      # p-Werte
> sum(pVals < alpha)                  # Anzahl signifikanter Tests
[1] 55
```

Während bei `for()` durch die Länge von `<Vektor>` festgelegt ist, wie häufig die Schleife durchlaufen wird, kann dies bei `while()` durch Berechnungen innerhalb der Schleife gesteuert werden.

```
> while(<Ausdruck>) { <Befehlsblock> }
```

Als Argument erwartet `while()` einen Ausdruck, der sich zu einem einzelnen Wahrheitswert `TRUE` oder `FALSE` auswerten lässt. Hierbei ist auszuschließen, dass der Ausdruck einen Vektor der Länge 0, `NA`, `NaN` oder `NULL` ergibt – etwa indem der Ausdruck in `isTRUE()` eingeschlossen wird (vgl. Abschn. 1.3.6).

Der in `{}` eingefasste Schleifenrumpf wird immer wieder durchlaufen, solange der Ausdruck gleich `TRUE` ist. Typischerweise ändern Berechnungen im Schleifenrumpf den Ausdruck, so dass dieser `FALSE` ergibt, sobald ein angestrebtes Ziel erreicht wird. Es ist zu gewährleisten, dass dies auch irgendwann der Fall ist, andernfalls handelt es sich um eine *Endlosschleife*, die den weiteren Programmablauf blockiert und auf der Konsole mit der `ESC` Taste abgebrochen werden müsste (unter Linux mit `Strg+C`).

```
> x <- 37
> y <- 10
> while(x >= y) { x <- x-y }      # Modulo-Berechnung (für positive Werte)
> x
[1] 7
```

Die Schlüsselwörter `break` und `next` erlauben es, die Ausführung von Schleifen innerhalb des Schleifenrumpfes zu steuern. Sie stehen i. d. R. innerhalb eines `if()` Blocks. Durch `break` wird die Ausführung der Schleife abgebrochen, noch ehe das hierfür im Schleifenkopf definierte Kriterium erreicht ist. Mit `next` bricht nur der aktuelle Schleifendurchlauf ab und geht zum nächsten Durchlauf über, ohne ggf. auf `next` folgende Befehle innerhalb des Schleifenrumpfes auszuführen. Die Schleife wird also fortgesetzt, die auf `next` folgenden Befehle dabei aber einmal übersprungen.

```
> for(i in 1:10) {
+   if((i %% 2) != 0) { next }
+   if((i %% 8) == 0) { break }
+   print(i)
+ }
[1] 2
[1] 4
[1] 6
```

Eine durch `repeat` eingeleitete Schleife wird solange ausgeführt, bis sie explizit durch `break` abgebrochen wird. Der Schleifenrumpf muss also eine Bedingung überprüfen und eine `break` Anweisung beinhalten, um eine Endlosschleife zu vermeiden.

```
> repeat { <Befehlsblock> }

> i <- 0
> repeat {
+   i <- i+1
+   if((i %% 4) == 0) { break }
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
```

15.2 Eigene Funktionen erstellen

Funktionen sind eine Zusammenfassung von Befehlen auf Basis von beim Aufruf mitgelieferten Eingangsinformationen, den *Funktionsargumenten*. Ebenso wie etwa Matrizen als Objekte der Klasse `matrix` können Funktionen als Objekte der Klasse `function` erstellt werden, indem über `function()` eine Funktion definiert und das Ergebnis einem Objekt zugewiesen wird. Selbst erstellte Funktionen haben denselben Status und dieselben Möglichkeiten wie mit R mitgelieferte Funktionen.

```
> <Name> <- function(<Argument1>=<Voreinstellung>, <Arg2>=<Voreinst.>, ...) {
+   <Befehlsblock>
+ }
```

Als Beispiel soll eine Funktion mit dem Namen `.First` erstellt werden, über die in der Datei `Rprofile.site` im `etc/` Ordner des R-Programmordners individuell festgelegt werden kann, welche Befehle zu Beginn jeder R-Sitzung automatisch auszuführen sind (vgl. Abschn. 1.2.2).

```
> .First <- function() {
+   library(car)                      # lade automatisch Paket car
+   print("Have a nice day!")          # gib Begrüßung aus
+ }
```

15.2.1 Funktionskopf

Zunächst ist bei einer Funktion der *Funktionskopf* innerhalb der runden Klammern () zu definieren, in dem durch Komma getrennt jene *formalen* Argumente benannt werden, die eine Funktion als Eingangsinformation akzeptiert. Die hier benannten Argumente stehen innerhalb

des Funktionsrumpfes (s. u.) als Objekte zur Verfügung.⁵ Auch ein leerer Funktionskopf ist möglich, wenn die folgenden Befehle nicht von äußereren Informationen abhängen.

Jedem formalen Argument kann auf ein Gleichheitszeichen folgend ein Wert zugewiesen werden, den das Argument als Voreinstellung (*default*) annimmt, sofern es beim Aufruf der Funktion nicht explizit genannt wird. Fehlt ein solcher default, muss beim Aufruf der Funktion zwingend ein Wert für das Argument übergeben werden. Argumente mit Voreinstellung sind beim Aufruf dagegen optional (vgl. Abschn. 1.2.5).

Argumente können von jeder Klasse, also auch ihrerseits Funktionen sein – eine Möglichkeit, die etwa bei `lapply()` oder `curve()` Verwendung findet. Eine solche Funktion höherer Ordnung, die eine Funktion als Argument akzeptiert und auf einen Wert abbildet, ist ein *Funktional*.

Anders als in vielen Programmiersprachen ist es nicht notwendig, im Funktionskopf explizit anzugeben, was für eine Klasse ein Objekt haben muss, das für ein Argument bestimmt ist. Im Funktionsrumpf sollte deshalb eine Prüfung erfolgen, ob beim Funktionsaufruf tatsächlich ein für die weiteren Berechnungen geeignetes Objekt für ein Argument übergeben wurde.

Das Argument ... besitzt eine besondere Bedeutung: Beim Aufruf der Funktion hierfür übergebene Werte können im Funktionsrumpf unter dem Namen ... verwendet werden. Dabei kann es sich um mehrere, durch Komma getrennte Werte handeln, die sich innerhalb der Funktion gemeinsam mittels `<- list(...)` in einem Objekt speichern und weitergeben lassen. Das Argument ... bietet sich besonders dann an, wenn im Funktionsrumpf eine Funktion verwendet wird, bei der noch nicht feststeht, ob und wenn ja wie viele Argumente sie ihrerseits benötigt, wenn die selbst definierte Funktion später ausgeführt wird.

15.2.2 Funktionsrumpf

Auf den Funktionskopf folgt eingeschlossen in geschweifte Klammern {} der sich ggf. über mehrere Zeilen erstreckende *Funktionsrumpf* als Block von Befehlen und durch # eingeleiteten Kommentaren. Findet die vollständige Funktionsdefinition in einer Zeile Platz, sind die geschweiften Klammern optional.

Die Befehle im Funktionsrumpf haben Zugriff auf alle übergebenen Argumente und auf die in einer R-Sitzung zuvor erstellten Objekte. Objekte, die innerhalb eines Funktionsrumpfes definiert werden, stehen dagegen nur innerhalb der Funktion zur Verfügung, verfallen also nach dem Aufruf der Funktion.⁶

Werden innerhalb des Funktionsrumpfes Argumente aus dem Funktionskopf verwendet, ist das *Lazy-Evaluation*-Prinzip zu beachten, nach dem der Inhalt von Funktionsargumenten erst mit ihrer ersten Verwendung ausgewertet wird.⁷ Im folgenden laute der Funktionskopf

⁵ Die aufgerufene Funktion kann nur Kopien der ursprünglichen Objekte, nicht aber die Objekte selbst ändern, da Argumente als *Wertparameter* (*call by value*) übergeben werden.

⁶ Dies sind *lokale* Objekte, sie existieren in einer beim Funktionsaufruf eigens erstellten Umgebung (vgl. Abschn. 1.3.1). Für das in diesem Kontext relevante, aber komplexe Thema der Regeln für die Gültigkeit von Objekten (*scoping*) vgl. Chambers (2008) und Ligges (2014).

⁷ Bis zur Auswertung ist das Argument ein *promise*. Für gewöhnliche Zuweisungen an Objekte gilt dagegen das *Eager-Evaluation*-Prinzip, nach dem der Inhalt von neuen Objekten schon bei der Zuweisung ausgewertet wird. Um auch für diese Objekte das Lazy-Evaluation-Prinzip zu nutzen, muss `delayedAssign()` verwendet werden.

`function(var1, var2=mean(var1))`. Wird nun beim Aufruf der Funktion für `var2` kein Wert übergeben, führt R die Berechnung `mean(var1)` als voreingestellten Wert für `var2` erst dann aus, wenn `var2` zum ersten Mal im Funktionsrumpf auftaucht. Jede in vorherigen Befehlen des Funktionsrumpfes ggf. vorgenommene Änderung an `var1` würde sich dann im Wert von `var2` niederschlagen. Die Funktionsrümpe { `var2` } und { `var1 <- var1^2; var2` } hätten damit unterschiedliche Ergebnisse für `var2` zur Folge. Bei der Verwendung von Zuweisungen in Voreinstellungen von Argumenten ist deshalb Sorgfalt geboten.

Das Lazy-Evaluation-Prinzip ermöglicht es auch, dass die Voreinstellung eines Arguments auf Objekte Bezug nimmt, die erst im Funktionsrumpf erstellt werden. Beispiel in Abschn. 15.2.3 sei eine Plot-Funktion mit einem optionalen Argument für den Wertebereich der Abszisse, wobei die Funktion bei Vektoren ungleicher Länge automatisch den längeren passend kürzt.

Häufig setzen Auswertungsschritte im Rumpf einer Funktion voraus, dass die als Argumente übergebenen Objekte von einer vorher festgelegten Struktur sind, es sich etwa um Matrizen einer bestimmten Dimensionierung, Vektoren von Zeichenketten, o. ä. handelt. Um sicherzustellen, dass erwartete Argumente auch tatsächlich beim Funktionsaufruf übergeben wurden und die richtige Form besitzen, ist es sinnvoll, den Funktionsrumpf mit entsprechenden Prüfungen zu beginnen.⁸ So prüft `missing("<Argumentname>")`, ob ein konkreter Wert für ein Argument übergeben wurde. Weiterhin sind Fallunterscheidungen mit `if()` oder Funktionen wie `is.<Klasse>()` (vgl. Abschn. 1.3) hierfür nützlich.

Die Ausgabe der im Funktionsrumpf beherbergten Befehle erscheint nicht auf der Konsole, die Arbeitsschritte der Funktion sind also beim Funktionsaufruf nicht sichtbar. Für Ausgaben auf der Konsole müssen innerhalb des Funktionsrumpfes deshalb die Funktionen `print()` und `cat()` verwendet werden (vgl. Abschn. 2.12.2).

15.2.3 Fehler behandeln

Entsprechen die einer Funktion übergebenen Argumente nicht den Anforderungen, kann die Funktion mit einer Fehlermeldung abgebrochen werden, indem `stop("<Fehlermeldung>")` ausgeführt wird. Soll die Funktion dagegen weiter ausgeführt und nur eine Warnmeldung ausgegeben werden, ist `warning("<Warnhinweis>")` zu verwenden. `stopifnot()` liefert eine andere Möglichkeit zum Abbruch einer Funktion, nachdem sie einen logischen Ausdruck geprüft hat.

```
> stopifnot(<logischer Ausdruck 1>, <logischer Ausdruck 2>, ...)
```

`stopifnot()` beendet die Funktion, sofern bereits einer der übergebenen Ausdrücke nicht `TRUE`, sondern `FALSE`, `NA`, `NULL` oder ein leerer Vektor ist und ersetzt damit eine Konstruktion wie etwa:

```
> if(!isTRUE(<logischer Ausdruck>)) { stop("<Fehlermeldung>") }
```

⁸Für einfache, nur zum eigenen Gebrauch bestimmte Funktionen mag dies überflüssig erscheinen, da man dann selbst darauf achten kann, sie korrekt zu verwenden. Allerdings ist dies gefährlich, da sich Funktionen durch andere als die vorgesehenen Eingangsinformationen stillschweigend anders als beabsichtigt verhalten können und so u. U. schwer entdeckbare Fehler verursachen (vgl. Abschn. 15.2.6).

Führt ein im Funktionsrumpf verwendeter Ausdruck zur Laufzeit zu einem Fehler, bricht die gesamte Funktion an dieser Stelle ab. Eine Funktion kann aber auch fehlertolerant werden, indem potentiell zu Fehlern führende Ausrücke in `try(<Ausdruck>)` eingeschlossen werden. Mit dieser Konstruktion ist die Funktion immun gegenüber vom Ausdruck produzierten Fehlern und setzt sich im Anschluss wie gewohnt fort. Schlägt der Ausdruck mit Fehlern fehl, gibt `try()` ein Objekt zurück, das aus der Klasse "try-error" abgeleitet ist. Mit `inherits(<try-Objekt>, →what="try-error")` lässt sich dies prüfen, um ggf. dann notwendige Befehle auszuführen (vgl. Abschn. 13.3 für ein Beispiel).

Um die Fehlerbehandlung mit `try()` auch bei Warnungen nutzen zu können, lässt sich mit `options(warn=2)` das Verhalten von R so ändern, dass Warnungen wie Fehler behandelt werden – Voreinstellung ist `options(warn=0)`.

Präziser lassen sich durch Ausdrücke im Funktionsrumpf ausgelöste Fehler und Warnungen mit `tryCatch()` behandeln.

```
> tryCatch({ <Ausdruck> },
+           error=function(e) { <Ausdruck> },
+           warning=function(w) { <Ausdruck> },
+           finally=         { <Ausdruck> })
```

Als erstes Argument ist der potentiell zu Fehlern oder Warnungen führende Ausdruck anzugeben – erstreckt er sich über mehrere Zeilen, ist er in `{ }` einzuschließen. Das Argument `error` erwartet eine Funktion, die ausgeführt wird, wenn es zu einem Fehler kommt. Sie erhält ihrerseits die Fehlermeldung als Argument. Analog ist für `warning` eine Funktion zu nennen, die bei Warnmeldungen aufgerufen werden soll. Ein für `finally` übergebener Ausdruck wird in jedem Fall im Anschluss an den eigentlichen Ausdruck sowie nach den für `error` und `warning` genannten Funktionen ausgewertet – auch wenn keine Fehler oder Warnungen auftreten.

```
# Voreinstellung für xLims ist ein erst im Rumpf erstelltes Objekt
> myPlot <- function(x, y, xLims=xRange) {
+   # stelle sicher, dass x und y numerisch und nicht leer sind
+   stopifnot(is.numeric(x), is.numeric(y))
+   stopifnot(length(x) > 0, length(y) > 0)
+
+   # prüfe, ob ein Vektor automatisch gekürzt werden muss
+   if(length(x) != length(y)) {
+     warning("x und y haben ungleiche Länge -> kürze")
+     lenMin <- min(length(x), length(y))
+     x <- x[1:lenMin]
+     y <- y[1:lenMin]
+   }
+
+   # fehlt xLims im Aufruf, verwende diesen Wertebereich für x
+   xRange <- round(range(x), -1) + c(-10, 10)
+
+   # Auswertung von Argument xLims erfolgt erst im nächsten Befehl
+   plot(x, y, xlim=xLims)
}
```

```
> myPlot(1:5, 1:10)
Warnmeldung:
In myPlot(1:5, 1:10) : x und y haben ungleiche Länge -> kürze
```

15.2.4 Rückgabewert und Funktionsende

Das von einer Funktion zurückgegebene Ergebnis ist die Ausgabe des letzten Befehls im Funktionsrumpf. Empfehlenswert ist es allerdings, die Rückgabe eines Objekts explizit über `return(<Objekt>)` erfolgen zu lassen. Dies beendet die Ausführung der Funktion, auch wenn im Funktionsrumpf weitere Auswertungsschritte folgen – eine Situation, die auftreten kann, wenn sich der Befehlsfluss unter Verwendung von Kontrollstrukturen verzweigt (vgl. Abschn. 15.1.1).

Mehrere Werte lassen sich gemeinsam zurückgeben, indem sie zuvor im Funktionsrumpf in einem geeigneten Objekt zusammengefasst werden, etwa einem Vektor, einer Matrix, einer Liste oder einem Datensatz.

Soll die Funktion keinen Wert zurückgeben, etwa weil sie nur Grafiken zu erstellen hat, ist als letzte Zeile der Befehl `return(invisible(NULL))` zu verwenden. `return(invisible(<Objekt>))` gibt ein Objekt zurück, das jedoch nach ihrem Aufruf nicht auf der Konsole sichtbar ist – wie dies etwa `barplot()` tut.

Mit `on.exit(<Ausdruck>)` kann zu Beginn im Funktionsrumpf ein Ausdruck festgelegt werden, der beim Beenden der Funktion ausgeführt wird – unabhängig davon, an welcher Position die Funktion tatsächlich verlassen wird. Dies ist etwa sinnvoll, wenn eine Funktion temporär globale Optionen mit `options()` ändern und am Schluss wieder auf den ursprünglichen Wert zurücksetzen soll (vgl. Abschn. 13.3 für ein Beispiel).

Da Funktionen selbst reguläre (*first class*) Objekte sind, lassen sie sich ebenfalls als Rückgabewert verwenden. Dabei ist die Besonderheit zu beachten, dass eine zurückgegebene Funktion eine Kopie der Daten der Umgebung mit einschließt, in der sie definiert wurde (*closure*). In der closure bleiben diese Daten konstant, auch wenn sie sich außerhalb der closure später ändern. Gemeinsam mit anderen Eigenschaften sind closures ein wesentliches Merkmal funktionaler Programmiersprachen Wickham (2014a). Weitere solche Merkmale sind die Möglichkeit, Funktionen höherer Ordnung (Funktionale, vgl. Abschn. 15.2.1) zu definieren, Funktionen in Listen zu speichern, anonyme Funktionen zu verwenden (vgl. Abschn. 15.2.5) und die Strategie, Funktionen so zu gestalten, dass sie keine *Seiteneffekte*, also Auswirkungen auf Objekte außerhalb der Funktion haben.

15.2.5 Eigene Funktionen verwenden

Nach ihrer Definition sind eigens erstellte Funktionen auf dieselbe Weise verwendbar wie die von R selbst oder von Zusatzpaketen zur Verfügung gestellten. Dies ist u. a. dann nützlich, wenn an Befehle wie `apply()` oder `replicate()` Funktionen übergeben werden können, vgl. etwa Abschn. 7.9.3, 10.1.2, 11.1, 14.8.1, oder 14.8.5 für Beispiele.

Funktionen müssen für ihre Verwendung nicht unbedingt in einem Objekt gespeichert werden. Analog zur Indizierung eines nicht als Objekt gespeicherten Vektors mit `c(1, 2, 3)[1]` lassen sich auch Funktionen direkt verwenden – man bezeichnet sie dann als *anonyme*, weil namenlose Funktionen. Diese lassen sich direkt innerhalb eines Funktionsaufrufs definieren, wo eine Funktion als Argument zu übergeben ist, vgl. etwa Abschn. 10.3.3 und 15.1.2 für Beispiele.

```
> (function(arg1, arg2) { arg1^2 + arg2^2 })(-3, 4)
[1] 25

# euklidische Länge jeder Spalte einer Matrix
> mat <- matrix(rnorm(16, 100, 15), nrow=4)
> apply(mat, 2, function(x) { sqrt(sum(x^2)) } )
[1] 218.9201 203.0049 205.1682 209.4517

> sqrt(colSums(mat^2))                                # Kontrolle: effizienter
[1] 218.9201 203.0049 205.1682 209.4517
```

15.2.6 Generische Funktionen

Generische Funktionen stellen eine Möglichkeit dar, mit der R objektorientiertes Programmieren unterstützt. Funktionen werden als generisch (oder auch als *polymorph*) bezeichnet, wenn sie sich abhängig von der Klasse der übergebenen Argumente automatisch unterschiedlich verhalten. Es existieren dann mehrere Varianten (*Methoden*) einer Funktion, die alle unter demselben allgemeinen Namen angesprochen werden können. Funktionen, für die mehrere Methoden existieren, werden auch als *überladen* bezeichnet. Die Methoden können ebenfalls unter einem eigenen, eindeutigen Namen aufgerufen werden, der nach dem Muster `<Funktionsname>.<Klasse>` aufgebaut ist, wobei sich `<Klasse>` auf die Klasse des ersten Funktionsarguments bezieht.⁹ So existiert etwa für `plot()` die Methode `plot.lm()` für den Fall, dass das erste Argument ein mit `lm()` angepasstes lineares Modell ist. Ebenso verhält sich z. B. `summary()` für numerische Vektoren anders als für Faktoren.

Die S3-Methoden einer Funktion nennt `methods("<Funktionsname>")`, auf die auch ihre Hilfe-Seite unter `Usage` verweist. Analog erhält man mit `methods(class="<Klasse>")` Auskunft darüber, welche Funktionen spezielle Methoden für Objekte einer bestimmten Klasse besitzen.¹⁰ Die generische Eigenschaft von Funktionen bleibt dem Anwender meist verborgen, da die richtige der unterschiedlichen Methoden automatisch in Abhängigkeit von der Klasse der übergebenen Argumente aufgerufen wird, ohne dass man dafür den passenden spezialisierten Funktionsnamen nennen müsste. Selbst erstellte Funktionen können im S3-Paradigma generisch gemacht werden, indem sie im Funktionsrumpf als letzten Befehl einen Aufruf von `UseMethod()` enthalten.

```
> UseMethod(generic="<Funktionsname>")
```

Für das Argument `generic` ist als Zeichenkette der Name der Funktion zu nennen, die generisch werden soll.

⁹Es handelt sich bei der hier vorgestellten Technik um das S3-Paradigma – in Abgrenzung zum flexibleren, aber auch komplizierteren S4-Paradigma, das etwa beim beschriebenen *method dispatch* nicht auf das erste Argument beschränkt ist.

¹⁰Für S4-Methoden analog `showMethods("<Funktionsname>")` sowie `showMethods(classes="<Klasse>")`.

In einem zweiten Schritt sind alle gewünschten Methoden der mit `generic` bezeichneten Funktion zu erstellen, deren Namen nach dem `<Funktionsname>.⟨Klasse⟩` Muster aufgebaut sein müssen. Für Argumente aller Klassen, die nicht explizit durch eine eigene Methode Berücksichtigung finden, muss eine Methode mit dem Namen `<Funktionsname>.default` angelegt werden, wenn die Funktion auch in diesem Fall arbeiten soll.

Als Beispiel wird eine Funktion `info()` erstellt, die eine wichtige Information über das übergebane Objekt ausgibt und dabei unterscheidet, ob es sich um einen numerischen Vektor (Klasse `numeric`), eine Matrix (Klasse `matrix`) oder einen Datensatz (Klasse `data.frame`) handelt.

```
> info <- function(x) { UseMethod("info") }      # generische Funktion

# Methoden für Objekte verschiedener Klassen
> info.numeric <- function(x) { range(x) }      # für numerische Vektoren
> info.matrix <- function(x) { dim(x) }          # für Matrizen
> info.data.frame <- function(x) { names(x) }     # für Datensätze
> info.default <- function(x) { length(x) }       # für andere Objekte

# Objekte verschiedener Klassen
> vec <- round(runif(12, 20, 100), 2)           # numerischer Vektor
> char <- LETTERS[sample(1:26, 5)]               # Vektor von Zeichenketten
> mat <- matrix(vec, nrow=3)                     # Matrix
> myDf <- data.frame(mat)                        # Datensatz
> names(myDf) <- LETTERS[1:ncol(myDf)]          # Variablennamen ändern

# Anwendung von info() auf Objekte verschiedener Klassen
> info(vec)                                     # ruft info.numeric() auf
[1] 26.05 91.62

> info(mat)                                     # ruft info.matrix() auf
[1] 3 4

> info(myDf)                                    # ruft info.data.frame() auf
[1] "A" "B" "C" "D"

> info(char)                                     # ruft info.default() auf
[1] 5
```

Die verschiedenen Methoden einer generischen Funktion können im Prinzip gänzlich andere Argumente als diese erwarten. In der Praxis empfiehlt es sich jedoch, diese Freiheit nur dahingehend zu nutzen, dass jede Methode zunächst alle Argumente der generischen Funktion in derselben Reihenfolge mit denselben Voreinstellungen verwendet. Zusätzlich können Methoden danach weitere Argumente besitzen, die sich je nach Methode unterscheiden.

15.3 Funktionen analysieren und verbessern

Ein wichtiger Bestandteil der vertieften Arbeit mit fremden und selbst erstellten Funktionen ist die Analyse der ausgeführten Befehlsabfolge, insbesondere hinsichtlich der Frage, ob die Funktion fehlerfrei und effizient die übertragenen Aufgaben umsetzt (*debugging*). Hierfür eignet sich zum einen die Begutachtung ihres Quelltextes, zum anderen die schrittweise Untersuchung ihres Verhaltens zur Laufzeit.¹¹

15.3.1 Quelltext fremder Funktionen begutachten

Aus welchen Befehlen sie besteht, ist bei selbst erstellten Funktionen bekannt. Bei fremden Funktionen hilft der Umstand, dass R den Quelltext auf der Konsole ausgibt, wenn der Funktionsname ohne runde Klammern als Befehl eingegeben wird.¹² Mit `capture.output(<Funktionsname>)` lässt sich dieser Quelltext in einem Vektor aus Zeichenketten zur späteren Analyse speichern. Bei `sd()` etwa ist erkennbar, dass letztlich `var()` aufgerufen und die Streuung als Wurzel der Varianz berechnet wird.

```
> sdSource <- capture.output(sd)
> sdSource
[1] "function (x, na.rm = FALSE) "
[2] "sqrt(var(if (is.vector(x)) x else as.double(x), na.rm = na.rm))"
[3] "<environment: namespace:stats>"
```

Um explizit auf Objekte einer bestimmten Umgebung (etwa der eines Pakets) zuzugreifen, ist dem Objektnamen der Paketname mit zwei Doppelpunkten voranzustellen. Alternativ eignet sich auch `get()`. Zusatzpakete können über *Namensräume (namespaces)* dafür sorgen, dass manche ihrer Objekte nur für Funktionen aus dem Paket selbst verwendbar sind, außerhalb davon jedoch nicht. Indem man den mit dem Paketnamen übereinstimmenden Namensraum mit drei Doppelpunkten dem Objektnamen voranstellt, lassen sich auch solche Objekte untersuchen.

```
> base::mean      # mean() aus Paket base ...
> boot:::basic.ci # basic.ci() aus Paket boot, sonst nicht sichtbar ...
```

Ist eine Funktion im S3-Paradigma generisch (vgl. Abschn. 15.2.6), enthält sie nur den `UseMethod()` Befehl. Über `methods("⟨Funktionsname⟩")` erfährt man jedoch, welche Methoden für eine solche Funktion existieren und sich über ihren vollständigen Namen ausgeben lassen.¹³ Manche Methoden sind dabei zunächst unsichtbar, was durch ein * Symbol deutlich gemacht wird, mit dem der Name der Methode in der Ausgabe von `methods()` versehen wird. `getS3method("⟨Funktionsname⟩", "⟨Klasse⟩")` hilft, auch den Inhalt solcher Methoden anzuzeigen.¹⁴ Demselben Zweck dient auch `getAnywhere("⟨Funktionsname⟩")`, wobei hier der vollständige Funktionsname i. S. von "⟨Basisname⟩.⟨Klasse⟩" anzugeben ist.

¹¹Die Entwicklungsumgebungen RStudio und Architect bzw. Eclipse+StatET bieten grafische Oberflächen, die das debugging sehr erleichtern.

¹²Operatoren sind in rückwärts gerichtete Hochkommata zu setzen, etwa `+` für die Addition.

¹³Für S4-Methoden analog `showMethods("⟨Funktionsname⟩")`.

¹⁴Für S4-Methoden analog `showMethods("⟨Funktionsname⟩", classes="⟨Klasse⟩", includeDefs=TRUE)`.

```

> get("fligner.test")                      # fligner.test ist generisch
function (x, ...)
UseMethod("fligner.test")
<environment: namespace:stats>

> methods("fligner.test")                  # Methoden für fligner.test
[1] fligner.test.default* fligner.test.formula*
Non-visible functions are asterisked

> get("fligner.test.default")             # Fehler: unsichtbare Methode
Fehler: Objekt 'fligner.test.default' nicht gefunden

> getS3method("fligner.test", "default")   # findet Methode ...
> getAnywhere("fligner.test.default")      # findet Methode ...

```

In jedem Fall ist es über den auf CRAN erhältlichen Quelltext von R und den der Zusatzpakete möglich, die Befehlsabfolge einer Funktion zu analysieren.

15.3.2 Funktionen zur Laufzeit untersuchen

Mit `cat()` und `print()` lassen sich innerhalb einer selbst geschriebenen Funktion Zwischenergebnisse auf der Konsole ausgeben, um während ihrer Entwicklung Anhaltspunkte darüber zu gewinnen, wie sich die Funktion zur Laufzeit verhält. Treten Fehler im Endergebnis auf, kann so womöglich die Ursache auf einen bestimmten Teilschritt eingegrenzt werden. Ebenso sollten Zwischenergebnisse, die von einem bestimmten Typ sein müssen, analog zu übergebenen Funktionsargumenten daraufhin geprüft werden, ob die Vorgaben auch eingehalten sind. Andernfalls ist die Funktion ggf. mit `stop("<Fehlermeldung>")` abzubrechen. `stopifnot()` stellt eine andere Möglichkeit dafür zur Verfügung, die das Prüfen eines logischen Ausdrucks bereits beinhaltet (vgl. Abschn. 15.2.2).

Treten Fehler beim Ausführen einer Funktion auf, zeigt `traceback()` den *call-stack* an, welche Funktionen also zuletzt ineinander verschachtelt aufgerufen wurden. Um diese Möglichkeit auch bei Warnungen nutzen zu können, lässt sich mit `options(warn=2)` das Verhalten von R so ändern, dass Warnungen wie Fehler behandelt werden. Damit bricht eine Funktion ab, sobald es zu einer Warnung kommt. Im Beispiel wird deutlich, dass zuletzt `stop()` ausgeführt und vorher aus der Funktion `stopifnot()` aufgerufen wurde, die wiederum innerhalb von `myPlot()` gestartet wurde (vgl. Abschn. 15.2.2).

```

> myPlot(1:2, "ABC")
Fehler: is.numeric(y) is not TRUE

> traceback()
3: stop(paste(ch, " is not ", if (length(r) > 1L) "all ", "TRUE",
   sep = ""), call. = FALSE)
2: stopifnot(is.numeric(x), is.numeric(y)) at selection.r#2
1: myPlot(1:2, "ABC")

```

Wenn durch den Aufruf von `traceback()` klar ist, in welcher Funktion ein Problem auftritt, kann man sich mit `debug(<Funktionsname>)` in diese direkt hinein begeben, während sie ausgeführt wird. Die zu analysierende Funktion ist als Argument zu übergeben. Dadurch wird sie zum debugging markiert und fortan bei ihrem Aufruf immer im *browser* ausgeführt, der zunächst alle Befehle der Funktion anzeigt und dann die schrittweise Abarbeitung jedes einzelnen Befehls ermöglicht. Jeder Ausdruck innerhalb der Funktion (eine Zeile oder ein durch `{ ... }` definierter Block mehrerer Zeilen) wird dabei zunächst separat angezeigt. Daraufhin ist `n (next)` bzw. die `Return` Taste zu drücken, um ihn auszuführen und zum nächsten Ausdruck zu gelangen.

Darüber hinaus kann im *browser* die Konsole von R wie gewohnt verwendet werden, um sich etwa mit `ls()`, `print()`, `head()` oder `str()` einen Überblick über den Inhalt der vorhandenen Objekte zu verschaffen und die verarbeiteten Daten zu begutachten. `c (continue)` setzt die Abarbeitung der Funktion ohne weitere Unterbrechung fort, `s (step into)` führt das debugging innerhalb der in der aktuellen Zeile auszuführenden Funktion weiter, `f (finish)` beendet den aktuellen Funktionsaufruf oder Schleifendurchlauf. Die Eingabe von `where` dient dazu, die Reihenfolge der durch die letzten ineinander verschachtelten Funktionsaufrufe erstellten Umgebungen zu nennen. Weitere Befehle gibt `help` im *browser* aus, `Q (quit)` bricht das debugging an der aktuellen Stelle ab.

Im *browser* kann eine Funktion über `fix(<Funktionsname>)` etwa zur Fehlerkorrektur geändert und dann erneut aufgerufen werden, um das neue Verhalten zu kontrollieren. Soll eine Funktion nicht weiter analysiert werden, ist ihr Name an `undebug(<Funktionsname>)` als Argument zu übergeben.

Als Beispiel diene die in Abschn. 15.2.6 erstellte generische `info()` Funktion.

```
> debug(info)                                # setze debug-flag
> info(sample(1:10, 10, replace=TRUE))       # Aufruf startet browser
debugging in: info(sample(1:10, 10, replace = TRUE))
debug: {
  UseMethod("info")
}

Browse[1]> n                                  # beginne Ausführung
debug: UseMethod("info")

Browse[1]> n                                  # führe 1. Befehl aus
debugging in: info.numeric(sample(1:10, 10, replace = TRUE))
debug: {
  range(x)
}

Browse[2]> ls()                               # vorhandene Objekte
[1] "x"

Browse[2]> x                                  # zeige Inhalt von x
[1] 5 8 2 4 8 9 7 9 6 2
```

```

Browse[1]> where                                # letzte Funktionsaufrufe
where 1: info.numeric(sample(1:10, 10, replace = TRUE))
where 2: info(sample(1:10, 10, replace = TRUE))

Browse[2]> c                                    # setze Funktion fort
exiting from: info.numeric(sample(1:10, 10, replace = TRUE))
exiting from: info(sample(1:10, 10, replace = TRUE))
[1] 2 9

> undebbug(info)                             # entferne debug-flag

```

Mitunter ist es mühsam, durch die schrittweise Ausführung einer Funktion im browser schließlich an die Stelle zu gelangen, die z. B. im Rahmen einer Fehlersuche als mögliche Problemquelle identifiziert wurde. Bei selbst erstellten Funktionen lässt sich der Debug-Prozess deswegen abkürzen: Mit Einfügen des `browser()` Befehls vor dem zu analysierenden Abschnitt setzt man einen Haltepunkt, durch den sich beim Ausführen der Funktion der browser öffnet, sobald die bezeichnete Stelle erreicht ist. Damit i. S. eines bedingten Haltepunkts nur dann das debugging gestartet wird, wenn bestimmte Randbedingungen gelten, ist `if(<Bedingung>) { browser() }` zu verwenden.

Für eine weitere Möglichkeit, Haltepunkte in selbst geschriebenen Skriptdateien zu setzen, vgl. `?setBreakpoint` sowie insbesondere die grafische Umsetzung innerhalb der Entwicklungs-umgebungen RStudio und Architect bzw. Eclipse+StatET. Bei fremden Funktionen dient die Option `options(error=recover)` dazu, beim Auftreten eines Fehlers direkt in den browser zu wechseln. Danach sollte die Option mit `options(error=NULL)` wieder zurückgesetzt werden.

15.3.3 Effizienz von Auswertungen steigern

Bei komplexeren Analysen und großen Datensätzen beginnt die Frage an Bedeutung zu gewinnen, wie die Geschwindigkeit der Auswertungsschritte erhöht werden kann. Bevor hierfür generelle Strategien vorgestellt werden, soll an die klassische Warnung vor einer verfrühten Optimierung erinnert werden: Wichtiger als besonders schnelle Berechnungen ist zunächst immer ihre Richtigkeit. Zudem setzt jede Optimierung eine eingehende Diagnostik voraus: Nur wenn bekannt ist, welche Einzelschritte einer Funktion besonders viel Zeit benötigen, weiß man, wo sinnvollerweise Zeit einzusparen ist. Hier hilft die Funktion `system.time(<Befehl>)`, die ausgibt, wieviel Rechenzeit die Abarbeitung eines Befehls benötigt hat (für eine Erweiterung vgl. das Paket `microbenchmark`; Mersmann, 2014). Weitere Hinweise zum *profiling* finden sich unter ?Rprof. Die Speichernutzung eines Objekts erfährt man durch `object.size(<Objekt>)`.

```

# Zeit, um (200x200)-Zufallsmatrix zu invertieren
> system.time(solve(matrix(sample(1:100, 200^2, replace=TRUE), nrow=200)))
User  System  verstrichen
0.06    0.00      0.06

# Speicherverbrauch double-Gleitkommazahlen: N*8 byte + overhead
> xDbl <- rnorm(10000, 0, 10)                  # Gleitkommazahlen
> object.size(xDbl)

```

```
80024 bytes
```

```
# Speicherverbrauch ganze Zahlen (integer): N*4 byte + overhead
> object.size(as.integer(xDb1))
40024 bytes
```

Bei der Analyse extrem großer Datenmengen besteht gegenwärtig das Problem, dass R Datensätze zur Bearbeitung im Arbeitsspeicher vorhalten muss, was die Größe von praktisch auswertbaren Datensätzen einschränkt (vgl. [?Memory](#)). Für Ansätze, diese Einschränkung zu umgehen vgl. den Abschnitt *High-Performance and Parallel Computing* der CRAN Task Views (Eddelbuettel, 2014). Er nennt auch Mittel zur besseren Ausnutzung paralleler Rechnerarchitekturen sowie Möglichkeiten, besonders zeitkritische Auswertungsschritte in kompilierte Sprachen wie C++ auszulagern (Eddelbuettel, 2013). Generell bieten die folgenden Ratschläge Potential, die Geschwindigkeit von Analysen zu erhöhen:

- Ein 64bit-System mit großzügig dimensioniertem Hauptspeicher samt angepasster maximaler Speichernutzung (vgl. [?Memory](#)) ist generell sinnvoll.
- Objekte schrittweise zu vergrößern, ist aufgrund der dafür notwendigen internen Kopiervorgänge ineffizient. Objekte sollten deshalb bereits mit der Größe angelegt werden, die sie später benötigen. Dabei ist der später benötigte Datentyp zu wählen – fügt man etwa einer vorher angelegten Matrix aus logischen Werten später eine Zahl hinzu, muss die gesamte Matrix per Kopie in einen numerischen Datentyp konvertiert werden (vgl. Abschn. 1.3.5).
- Bei großen Datensätzen ist nach Möglichkeiten zu suchen, nur mit Teilmengen der Daten zu arbeiten und ggf. eine Datenbank als Speicherort zu nutzen. Über eine lokale Datenbankanbindung können Daten meist schneller als mittels `read.table()` eingelesen werden. Für Daten in Textform aus sehr großen Dateien arbeitet `fread()` aus dem Paket `data.table` ebenfalls deutlich schneller als `read.table()`.
- Ganzzahlige Werte können mit `as.integer()` als solche gespeichert werden und benötigen dadurch weniger Speicher als die normalen Gleitkommazahlen doppelter Genauigkeit (vgl. Abschn. 1.3.5, Fußnote 24).
- Werden sehr große Objekte nicht mehr benötigt, sollten sie mit `rm()` entfernt werden, dabei ist auch an das automatisch erzeugte Objekt `.Last.value` zu denken.
- Objekte ohne Namensattribut (z. B. im Fall von Vektoren oder Matrizen) werden schneller als solche mit Namen verarbeitet. Zudem bieten Funktionen wie `lapply()`, `sapply()` oder `mapply()` an, Namensattribute mit dem Argument `USE.NAMES=FALSE` unberücksichtigt zu lassen.
- Matrizen sind effizienter zu verarbeiten als Datensätze.
- Bei vielen Auswertungen kommen intern Matrix-Operationen zum Einsatz (z. B. Singulärwertzerlegung, vgl. Abschn. 12.1), die bei sehr großen Datenmengen die Auswertungszeit entscheidend bestimmen können. Diese Rechnungen können von einer für den im Computer verwendeten Prozessor optimierten Version der *BLAS*-Bibliothek (basic linear algebra subprograms, vgl. Frage 8.2 in Ripley & Murdoch, 2014) profitieren.

- Schleifen sind in R als interpretierter und nicht kompilierter Sprache ein eher ineffizientes Sprachkonstrukt. Nach Möglichkeit sollte deshalb die für R typische vektorwertige Formulierung von Rechnungen gewählt werden, die meist schneller ist (Ligges & Fox, 2008).
- Während `apply()` in seiner Verwendung parallelisiert scheint, beruht es tatsächlich auf R-Schleifen. Dagegen greift `lapply()` (und damit `sapply()`) intern auf kompilierten Code zurück, was für einen Geschwindigkeitsvorteil sorgen kann.
- Das im Basisumfang von R enthaltene Paket `compiler` kann mit `cmpfun(<Funktion>)` eine Funktion zu *byte-code* kompilieren. Das Ergebnis ist eine Funktion, die zur ursprünglichen äquivalent ist, jedoch moderate Geschwindigkeitsvorteile besitzen kann.

Literaturverzeichnis

- Adler, D. & Murdoch, D. (2014). rgl: 3D visualization device system (OpenGL) [Software]. URL <http://CRAN.R-project.org/package=rgl> (R package version 0.93.1143)
- Adler, J. (2012). *R in a Nutshell* (2. Aufl.). Sebastopol, CA: O'Reilly.
- Agresti, A. (2007). *An Introduction to Categorical Data Analysis* (2. Aufl.). New York, NY: Wiley.
- Aiken, L. S. & West, S. G. (1991). *Multiple regression: Testing and interpreting interactions*. Thousand Oaks, CA: Sage.
- Allignol, A. & Latouche, A. (2014). CRAN task view: Survival analysis. URL <http://CRAN.R-project.org/view=Survival> (Version 2014-09-05)
- Andres, J. (1996). Das allgemeine lineare Modell. In E. Erdfelder, R. Mausfeld, T. Meiser & G. Rudinger (Hrsg.), *Handbuch Quantitative Methoden* (S. 185–200). Weinheim: Beltz.
- Andrews, F. (2012). playwith: A GUI for interactive plots using GTK+ [Software]. URL <http://CRAN.R-project.org/package=playwith> (R package version 0.9-54)
- Backhaus, K., Erichson, B., Plinke, W. & Weiber, R. (2011). *Multivariate Analysemethoden* (13. Aufl.). Heidelberg: Springer.
- Bates, D. (2004). Least Squares Calculations in R. *R News*, 4 (1), 17–20. URL <http://CRAN.R-project.org/doc/Rnews/>
- Bates, D. & Maechler, M. (2014). Matrix: Sparse and Dense Matrix Classes and Methods [Software]. URL <http://CRAN.R-project.org/package=Matrix> (R package version 1.1-4)
- Bates, D., Maechler, M., Bolker, B. & Walker, S. (2014). lme4: Linear mixed-effects models using S4 classes [Software]. URL <https://github.com/lme4/lme4/> (R package version 1.1-7)
- Becker, R. A., Chambers, J. M. & Wilks, A. R. (1988). *The New S language: A Programming Environment for Data Analysis and Graphics*. Pacific Grove, CA: Wadsworth & Brooks/Cole.
- Bender, R., Augustin, T. & Blettner, M. (2005). Generating survival times to simulate Cox proportional hazards models. *Statistics in Medicine*, 24, 1713–1723.
- Bernaards, C. A. & Jennrich, R. I. (2005). Gradient Projection Algorithms and Software for Arbitrary Rotation Criteria in Factor Analysis. *Educational and Psychological Measurement*, 65 (5), 676–696. URL <http://www.stat.ucla.edu/research/gpa/>
- Bliese, P. (2013). multilevel: Multilevel Functions [Software]. URL <http://CRAN.R-project.org/package=multilevel> (R package version 2.5)
- Boker, S., Neale, M., Maes, H., Wilde, M., Spiegel, M., Brick, T., ... Fox, J. (2011). OpenMx: An Open Source Extended Structural Equation Modeling Framework. *Psychometrika*, 66 (2), 306–317. URL <http://openmx.psyc.virginia.edu/>
- Borchers, H. W. (2014). pracma: Practical Numerical Math Functions [Software]. URL <http://CRAN.R-project.org/package=pracma> (R package version 1.7.7)
- Bortz, J., Lienert, G. A. & Boehnke, K. (2010). *Verteilungsfreie Methoden in der Biostatistik* (3. Aufl.). Heidelberg: Springer.

- Bronstein, I. N., Semendjajew, K. A., Musiol, G. & Mühlig, H. (2012). *Taschenbuch der Mathematik* (8. Aufl.). Frankfurt am Main: Harri Deutsch.
- Büning, H. & Trenkler, G. (1994). *Nichtparametrische statistische Methoden* (2. Aufl.). Berlin: Walter de Gruyter.
- Canty, A. & Ripley, B. D. (2014). boot: Bootstrap R (S-Plus) Functions [Software]. URL <http://CRAN.R-project.org/package=boot> (R package version 1.3-13)
- Carr, D., Lewin-Koh, N. & Maechler, M. (2014). hexbin: Hexagonal Binning Routines [Software]. URL <http://CRAN.R-project.org/package=hexbin> (R package version 1.27.0)
- Chamberlain, S., Ram, K., Gandrud, C. & Mair, P. (2014). CRAN task view: Web technologies and services. URL <http://CRAN.R-project.org/view=WebTechnologies> (Version 2014-08-29)
- Chambers, J. M. (2008). *Software for Data Analysis: Programming with R*. New York, NY: Springer. URL <http://statweb.stanford.edu/~jmc4/Rbook/>
- Champely, S. (2012). pwr: Basic functions for power analysis [Software]. URL <http://CRAN.R-project.org/package=pwr> (R package version 1.1.1)
- Chang, W. (2013). *R Graphics Cookbook*. Sebastopol, CA: O'Reilly. URL <http://www.cookbook-r.com/>
- Chernik, M. R. & LaBudde, R. A. (2011). *An Introduction to Bootstrap Methods with Applications to R*. Hoboken, NJ: Wiley.
- Chihara, L. & Hesterberg, T. (2011). *Mathematical Statistics with Resampling and R*. Hoboken, NJ: Wiley. URL <https://sites.google.com/site/chiharahesterberg/>
- Circle Systems. (2014). Stat/Transfer [Software]. URL <http://www.stattransfer.com/> (Version 12)
- Cowlishaw, M. F. (2008). Decimal Arithmetic FAQ Part 1 – General Questions. URL <http://speleotrove.com/decimal/decifaq1.html>
- Dalgaard, P. (2007). New Functions for Multivariate Analysis. *R News*, 7 (2), 2–7. URL <http://CRAN.R-project.org/doc/Rnews/>
- Dalgaard, P. (2008). *Introductory Statistics with R* (2. Aufl.). London, UK: Springer. URL <http://www.biostat.ku.dk/~pd/ISwR.html>
- Davison, A. C. & Hinkley, D. V. (1997). *Bootstrap Methods and Their Applications*. Cambridge, UK: Cambridge University Press.
- De Rosario-Martinez, H. (2013). phia: Post-hoc interaction analysis [Software]. URL <http://CRAN.R-project.org/package=phia> (R package version 0.1-5)
- Dowle, M., Short, T., Lianoglou, S. & Srinivasan, A. (2014). data.table: Extension of data.frame [Software]. URL <http://CRAN.R-project.org/package=data.table> (R package version 1.9.4)
- Dutang, C. (2014). CRAN task view: Probability distributions. URL <http://CRAN.R-project.org/view=Distributions> (Version 2014-08-19)
- Eddelbuettel, D. (2013). *Seamless R and C++ Integration with Rcpp*. New York, NY: Springer.
- Eddelbuettel, D. (2014). CRAN task view: High-performance and parallel computing with R. URL <http://CRAN.R-project.org/view=HighPerformanceComputing> (Version 2014-09-05)
- Eid, M., Gollwitzer, M. & Schmitt, M. (2013). *Statistik und Forschungsmethoden* (3. Aufl.). Weinheim: Beltz.
- Everitt, B. S. & Hothorn, T. (2010). *A Handbook of Statistical Analysis Using R* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC.

- Everitt, B. S. & Hothorn, T. (2011). *An Introduction to Applied Multivariate Analysis with R*. New York, NY: Springer.
- Everitt, B. S. & Hothorn, T. (2014). HSAUR2: A Handbook of Statistical Analyses Using R (2nd Edition) [Software]. URL <http://CRAN.R-project.org/package=HSAUR2> (R package version 1.1-11)
- Faraway, J. J. (2014). *Linear Models with R* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC. URL <http://www.maths.bath.ac.uk/~jjf23/LMR/>
- Faria, J. C. (2014). Resources of Tinn-R GUI/Editor for R Environment [Software]. Ilheus, Bahia, Brasil. URL <http://sourceforge.net/projects/tinn-r/> (Version 3.0.3.6)
- Faul, F., Erdfelder, E., Lang, A.-G. & Buchner, A. (2007). G*Power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior Research Methods*, 39 (2), 175–191. URL <http://www.gpower.hhu.de/>
- Fellows, I. (2014). Deducer [Software]. URL <http://CRAN.R-project.org/package=Deducer> (R package version 0.7-7)
- Filzmoser, P., Fritz, H. & Kalcher, K. (2014). pcaPP: Robust PCA by Projection Pursuit [Software]. URL <http://CRAN.R-project.org/package=pcaPP> (R package version 1.9-60)
- Filzmoser, P. & Gschwandtner, M. (2014). mvoutlier: Multivariate outlier detection based on robust methods [Software]. URL <http://CRAN.R-project.org/package=mvoutlier> (R package version 2.0.5)
- Fischer, G. (2010). *Lineare Algebra* (17. Aufl.). Wiesbaden: Vieweg+Teubner.
- Fox, J. (2003). Effect Displays in R for Generalised Linear Models. *Journal of Statistical Software*, 8 (15), 1–27. URL <http://www.jstatsoft.org/v08/i15/>
- Fox, J. (2005). The R Commander: A basic-statistics graphical user interface to R. *Journal of Statistical Software*, 14 (9), 1–42. URL <http://www.jstatsoft.org/v14/i09/>
- Fox, J. (2010). polycor: Polychoric and Polyserial Correlations [Software]. URL <http://CRAN.R-project.org/package=polycor> (R package version 0.7-8)
- Fox, J. (2014). CRAN task view: Statistics for the social sciences. URL <http://CRAN.R-project.org/view=SocialSciences> (Version 2014-08-08)
- Fox, J., Nie, Z. & Byrnes, J. (2014). sem: Structural Equation Models [Software]. URL <http://CRAN.R-project.org/package=sem> (R package version 3.1-5)
- Fox, J. & Weisberg, S. (2011). *An R Companion to Applied Regression* (2. Aufl.). Thousand Oaks, CA: Sage. URL <http://socserv.socsci.mcmaster.ca/jfox/Books/Companion/>
- Fox, J. & Weisberg, S. (2014). car: Companion to Applied Regression [Software]. URL <http://CRAN.R-project.org/package=car> (R package version 2.0-21)
- Friedl, J. E. F. (2006). *Mastering Regular Expressions* (3. Aufl.). Sebastopol, CA: O'Reilly. URL <http://regex.info/>
- Friedman, J., Hastie, T. & Tibshirani, R. (2010). Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software*, 33 (1), 1–22. URL <http://www.jstatsoft.org/v33/i01/>
- Gałecki, A. T. & Burzykowski, T. (2013). *Linear Mixed-Effects Models: A Step-by-Step Approach*. New York, NY: Springer.
- Gandrud, C. (2014). *Reproducible research with R & RStudio*. Boca Raton, FL: Chapman & Hall/CRC. URL <http://christophergandrud.github.io/RepResR-RStudio/>
- Gentleman, R. C., Carey, V. J., Bates, D. M. & others. (2004). Bioconductor: Open software development for computational biology and bioinformatics. *Genome Biology*, 5, R80. URL <http://genomebiology.com/2004/5/10/R80>

- Genz, A. & Bretz, F. (2009). *Computation of Multivariate Normal and t Probabilities. Lecture Notes in Statistics Vol. 195*. Heidelberg: Springer.
- Genz, A., Bretz, F., Miwa, T., Mi, X., Leisch, F., Scheipl, F. & Hothorn, T. (2014). mvtnorm: Multivariate Normal and t Distributions [Software]. URL <http://CRAN.R-project.org/package=mvtnorm> (R package version 1.0-0)
- Gesmann, M. & de Castillo, D. (2011). googleVis: Interface between R and the Google Visualisation API. *The R Journal*, 3 (2), 40–44. URL <http://journal.r-project.org/archive/2011-2/>
- Goldberg, D. (1991). What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23 (1), 5–48. URL <http://www.validlab.com/goldberg/paper.pdf>
- Good, P. I. (2004). *Permutation, Parametric, and Bootstrap Tests of Hypotheses* (3. Aufl.). New York, NY: Springer.
- Goulet, V., Dutang, C., Maechler, M., Firth, D., Shapira, M. & Stadelmann, M. (2014). expm: Matrix exponential [Software]. URL <http://CRAN.R-project.org/package=expm> (R package version 0.99-1.1)
- Goyaerts, J. & Levithan, S. (2012). *Regular expressions cookbook*. Sebastopol, CA: O'Reilly. URL <http://www.regular-expressions.info/>
- Graves, S., Dorai-Raj, S. & Francois, R. (2013). sos: Search contributed R packages, sort by package [Software]. URL <http://CRAN.R-project.org/package=sos> (R package version 1.3-8)
- Grolemund, G. & Wickham, H. (2011). Dates and Times Made Easy with lubridate. *Journal of Statistical Software*, 40 (3), 1–25. URL <http://www.jstatsoft.org/v40/i03/>
- Grothendieck, G. & Petzoldt, T. (2004). Date and Time Classes in R. *R News*, 4 (1), 29–32. URL <http://CRAN.R-project.org/doc/Rnews/>
- Härdle, W. K. & Simar, L. (2012). *Applied Multivariate Statistical Analysis* (3. Aufl.). Berlin: Springer.
- Harrell Jr, F. E. (2014a). Hmisc: Harrell Miscellaneous [Software]. URL <http://CRAN.R-project.org/package=Hmisc> (R package version 3.14-4)
- Harrell Jr, F. E. (2014b). rms: Regression Modeling Strategies [Software]. URL <http://CRAN.R-project.org/package=rms> (R package version 4.2-1)
- Harrell Jr, F. E. (2015). *Regression Modeling Strategies* (2. Aufl.). New York: Springer. URL <http://biostat.mc.vanderbilt.edu/wiki/Main/RmS>
- Hastie, T., Tibshirani, R. & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (2. Aufl.). New York: Springer. URL <http://statweb.stanford.edu/~tibs/ElemStatLearn/>
- Heiberger, R. M. & Neuwirth, E. (2009). *R Through Excel*. New York, NY: Springer.
- Heinze, G., Ploner, M., Dunkler, D. & Southworth, H. (2013). logistf: Firth's bias reduced logistic regression [Software]. URL <http://CRAN.R-project.org/package=logistf> (R package version 1.21)
- Helbig, M., Urbanek, S. & Fellows, I. (2013). JGR: Java Gui for R [Software]. URL <http://CRAN.R-project.org/package=JGR> (R package version 1.7-16)
- Hendrickx, J. (2012). perturb: Tools for evaluating collinearity [Software]. URL <http://CRAN.R-project.org/package=perturb> (R package version 2.05)
- Hesterberg, T. (2014). resample: Resampling functions [Software]. URL <http://CRAN.R-project.org/package=resample> (R package version 0.2)

- Hewson, P. (2014). *CRAN task view: Multivariate statistics*. URL <http://CRAN.R-project.org/view=Multivariate> (Version 2014-07-29)
- Hoffman, D. D. (2000). *Visual Intelligence: How We Create What We See*. New York, NY: W. W. Norton & Company.
- Højsgaard, S., Halekoh, U. & Yan, J. (2006). The R package geepack for generalized estimating equations. *Journal of Statistical Software*, 15 (2), 1–11. URL <http://www.jstatsoft.org/v15/i02/>
- Honaker, J., King, G. & Blackwell, M. (2011). Amelia: II: A Program for Missing Data. *Journal of Statistical Software*, 45 (7), 1–47. URL <http://www.jstatsoft.org/v45/i07/>
- Hornik, K. (2014). The R FAQ [Software-Handbuch]. URL <http://CRAN.R-project.org/doc/FAQ/>
- Hosmer Jr, D. W., Lemeshow, S. & May, S. (2008). *Applied survival analysis: Regression modeling of time to event data* (2. Aufl.). New York, NY: Wiley.
- Hothorn, T. (2014). *CRAN task view: Machine learning & statistical learning*. URL <http://CRAN.R-project.org/view=MachineLearning> (Version 2014-08-30)
- Hothorn, T., Bretz, F. & Westfall, P. (2008). Simultaneous Inference in General Parametric Models. *Biometrical Journal*, 50 (3), 346–363.
- Hothorn, T., Hornik, K., van de Wiel, M. A. & Zeileis, A. (2008). Implementing a Class of Permutation Tests: The coin Package. *Journal of Statistical Software*, 28 (8), 1–23. URL <http://www.jstatsoft.org/v28/i08/>
- Hyndman, R. J. (2014). *CRAN task view: Time series analysis*. URL <http://CRAN.R-project.org/view=TimeSeries> (Version 2014-08-26)
- Ihaka, R. & Gentleman, R. (1996). R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5 (3), 299–314.
- Ihaka, R., Murrell, P., Fisher, J. C. & Zeileis, A. (2013). colorspace: Color Space Manipulation [Software]. URL <http://CRAN.R-project.org/package=colorspace> (R package version 1.2-4)
- James, G., Witten, D., Hastie, T. & Tibshirani, R. (2013). *An introduction to statistical learning with applications in R*. New York, NY: Springer. URL <http://www-bcf.usc.edu/~gareth/ISL/>
- Johnson, P. E. (2014). rockchalk: Regression Estimation and Presentation [Software]. URL <http://CRAN.R-project.org/package=rockchalk> (R package version 1.8.86)
- Kelley, K. & Lai, K. (2012). MBESS [Software]. URL <http://CRAN.R-project.org/package=MBESS> (R package version 3.3.3)
- Kirk, R. E. (2013). *Experimental Design – Procedures for the Social Sciences* (4. Aufl.). Thousand Oaks, CA: SAGE. URL <http://www.sagepub.com/kirk/>
- Klein, J. P. & Moeschberger, M. L. (2003). *Survival analysis: Techniques for censored and truncated data* (2. Aufl.). New York, NY: Springer.
- Kuhn, M. (2014). caret: Classification and regression training [Software]. URL <http://CRAN.R-project.org/package=caret> (R package version 6.0-35)
- Kuhn, M. & Johnson, K. (2013). *Applied predictive modeling*. New York, NY: Springer. URL <http://appliedpredictivemodeling.com/>
- Leisch, F. & Gruen, B. (2014). *CRAN task view: Cluster analysis & finite mixture models*. URL <http://CRAN.R-project.org/view=Cluster> (Version 2014-09-05)
- Lemon, J. (2006). Plotrix: a package in the red light district of R. *R News*, 6 (4), 8–12.
- Ligges, U. (2002). R Help Desk: Automation of Mathematical Annotation in Plots. *R News*, 2 (3), 32–34. URL <http://CRAN.R-project.org/doc/Rnews/>

- Ligges, U. (2003). R Help Desk: Package Management. *R News*, 3 (3), 37–39. URL <http://CRAN.R-project.org/doc/Rnews/>
- Ligges, U. (2014). *Programmieren mit R* (4. Aufl.). Berlin: Springer Spektrum. URL <http://www.statistik.tu-dortmund.de/~ligges/PmitR/>
- Ligges, U. & Fox, J. (2008). R Help Desk: How can I avoid this loop or make it faster? *R News*, 8 (1), 46–50. URL <http://CRAN.R-project.org/doc/Rnews/>
- Long, J. D. (2012). *Longitudinal Data Analysis for the Behavioral Sciences Using R*. Thousand Oaks, CA: Sage. URL <http://www.sagepub.com/long/>
- Lumley, T. (2009). leaps: Regression subset selection (using Fortran code by Alan Miller) [Software]. URL <http://CRAN.R-project.org/package=leaps> (R package version 2.9)
- Maechler, M. (2014). CRAN task view: Robust statistical methods. URL <http://CRAN.R-project.org/view=Robust> (Version 2014-06-08)
- Maindonald, J. & Braun, W. J. (2010). *Data Analysis and Graphics Using R: An Example-Based Approach* (3. Aufl.). Cambridge, UK: Cambridge University Press. URL <http://maths-people.anu.edu.au/~johnm/r-book/daagur3.html>
- Maindonald, J. & Braun, W. J. (2014). DAAG: Data Analysis And Graphics data and functions [Software]. URL <http://CRAN.R-project.org/package=DAAG> (R package version 1.20)
- Mair, P. (2014). CRAN task view: Psychometric models and methods. URL <http://CRAN.R-project.org/view=Psychometrics> (Version 2014-08-08)
- Mardia, K. V., Kent, J. T. & Bibby, J. M. (1980). *Multivariate Analysis*. London, UK: Academic Press.
- Maronna, R. A., Martin, R. D. & Yohai, V. J. (2006). *Robust Statistics: Theory and Methods*. New York, NY: Wiley.
- Maxwell, S. E. & Delaney, H. D. (2004). *Designing experiments and analyzing data: A model comparison perspective* (2. Aufl.). Mahwah, NJ: Lawrence Erlbaum.
- Mazerolle, M. J. (2014). AICcmodavg: Model selection and multimodel inference based on (Q)AIC(c) [Software]. URL <http://CRAN.R-project.org/package=AICcmodavg> (R package version 2.0-1)
- Mersmann, O. (2014). microbenchmark: Sub microsecond accurate timing functions. [Software]. URL <http://CRAN.R-project.org/package=microbenchmark> (R package version 1.4-2)
- Meyer, D. & Hornik, K. (2009). Generalized and Customizable Sets in R. *Journal of Statistical Software*, 31 (2), 1–27. URL <http://www.jstatsoft.org/v31/i02/>
- Meyer, D., Zeileis, A. & Hornik, K. (2014). vcd: Visualizing Categorical Data [Software]. URL <http://CRAN.R-project.org/package=vcd> (R package version 1.3-2)
- Miller, A. J. (2002). *Subset selection in regression* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC.
- Mirai Solutions GmbH. (2014). XLConnect: Excel Connector for R [Software]. URL <http://CRAN.R-project.org/package=XLConnect> (R package version 0.2-9)
- Muenchen, R. A. (2011). *R for SAS and SPSS Users* (2. Aufl.). New York, NY: Springer. URL <http://r4stats.com/>
- Muenchen, R. A. & Hilbe, J. M. (2010). *R for Stata Users*. New York, NY: Springer. URL <http://r4stats.com/>
- Murrell, P. (2011). *R Graphics* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC. URL <http://www.stat.auckland.ac.nz/~paul/RG2e/>

- Neuwirth, E. (2011). RColorBrewer: ColorBrewer palettes [Software]. URL <http://CRAN.R-project.org/package=RColorBrewer> (Version 1.0-5)
- Neuwirth, E. (2014). RExcel [Software]. URL <http://rcom.univie.ac.at/> (Version 3.2.14)
- Oksanen, J., Blanchet, F. G., Kindt, R., Legendre, P., Minchin, P. R., O'Hara, R. B., ... Wagner, H. (2014). vegan: Community Ecology Package [Software]. URL <http://CRAN.R-project.org/package=vegan> (R package version 2.2-0)
- OpenAnalytics BVBA. (2014). Architect [Software]. URL <http://www.openanalytics.eu/architect/> (Version 0.9.4)
- Pau, G., Fuchs, F., Sklyar, O., Boutros, M. & Huber, W. (2010). EBImage – an R package for image processing with applications to cellular phenotypes. *Bioinformatics*, 26 (7), 979–981. URL <http://www.bioconductor.org/packages/devel/bioc/html/EBImage.html>
- Peters, A. & Hothorn, T. (2013). ipred: Improved predictors [Software]. URL <http://CRAN.R-project.org/package=ipred> (R package version 0.9-3)
- Pinheiro, J. C. & Bates, D. M. (2000). *Mixed-Effects Models in S and S-PLUS*. New York, NY: Springer.
- Pinheiro, J. C., Bates, D. M., DebRoy, S., Sarkar, D. & R Core Team. (2014). nlme: Linear and Nonlinear Mixed Effects Models [Software]. URL <http://CRAN.R-project.org/package=nlme> (R package version 3.1-118)
- Plate, T. & Heiberger, R. (2011). abind: Combine multi-dimensional arrays [Software]. URL <http://CRAN.R-project.org/package=abind> (R package version 1.4-0)
- Ploner, M. & Heinze, G. (2013). coxphf: Cox regression with Firth's penalized likelihood [Software]. URL <http://CRAN.R-project.org/package=coxphf> (R package version 1.10)
- Polzehl, J. & Tabelow, K. (2007). Adaptive Smoothing of Digital images: The R Package adimpro. *Journal of Statistical Software*, 19 (1), 1–17. URL <http://www.jstatsoft.org/v19/i01/>
- Poncet, P. (2012). modeest: Mode Estimation [Software]. URL <http://CRAN.R-project.org/package=modeest> (R package version 2.1)
- R Core Members, DebRoy, S., Bivand, R. et al. (2014). foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, ... [Software]. URL <http://CRAN.R-project.org/package=foreign> (R package version 0.8-61)
- R Development Core Team. (2014a). R: A Language and Environment for Statistical Computing [Software-Handbuch]. Vienna, Austria. URL <http://www.r-project.org/>
- R Development Core Team. (2014b). R: Data Import/Export [Software-Handbuch]. Vienna, Austria. URL <http://CRAN.R-project.org/doc/manuals/R-data.html>
- R Development Core Team. (2014c). R: Installation and Administration [Software-Handbuch]. Vienna, Austria. URL <http://CRAN.R-project.org/doc/manuals/R-admin.html>
- R Development Core Team. (2014d). Writing R Extensions [Software-Handbuch]. Vienna, Austria. URL <http://cran.r-project.org/doc/manuals/R-exts.html>
- R Foundation for Statistical Computing. (2013). R: Regulatory compliance and validation issues – a guidance document for the use of R in regulated clinical trial environments [Software-Handbuch]. Vienna, Austria. URL <http://www.r-project.org/doc/R-FDA.pdf>
- R Special Interest Group on Databases. (2014). DBI: R database interface [Software]. URL <http://CRAN.R-project.org/package=DBI> (R package version 0.3.1)
- Revelle, W. (2014). psych: Procedures for Psychological, Psychometric, and Personality Research [Software]. URL <http://CRAN.R-project.org/package=psych> (R package version 1.4.8.11)

- Revolution Analytics. (2014). Revolution R enterprise [Software]. URL <http://www.revolution-computing.com/> (Version 7)
- Ripley, B. D. (2013). RODBC: ODBC Database Access [Software]. URL <http://CRAN.R-project.org/package=RODBC> (R package version 1.3-9)
- Ripley, B. D. & Murdoch, D. (2014). *R for Windows FAQ*. URL <http://CRAN.R-project.org/bin/windows/base/rw-FAQ.html>
- Ritz, C. & Streibig, J. C. (2009). *Nonlinear Regression with R*. New York, NY: Springer.
- Robin, X., Turck, N., Hainard, A., Tiberti, N., Lisacek, F., Sanchez, J.-C. & Müller, M. (2011). pROC: an open-source package for R and S+ to analyze and compare ROC curves. *BMC Bioinformatics*, 12, 77. URL <http://web.expasy.org/pROC/>
- Rosseel, Y. (2012). lavaan: An R Package for Structural Equation Modeling. *Journal of Statistical Software*, 48 (2), 1–36. URL <http://www.jstatsoft.org/v48/i02/>
- Rossini, A. J., Maechler, M., Sparapani, R. A., Eglen, S., Spinu, V. & Heiberger, R. M. (2014). ESS: Emacs Speaks Statistics [Software]. URL <http://ess.r-project.org/> (Version 14.09)
- Rousseeuw, P. J., Croux, C., Todorov, V., Ruckstuhl, A., Salibian-Barrera, M., Verbeke, T. & Maechler, M. (2014). robustbase: Basic Robust Statistics [Software]. URL <http://CRAN.R-project.org/package=robustbase> (R package version 0.91-1)
- Rousseeuw, P. J. & van Zomeren, B. C. (1990). Unmasking Multivariate Outliers and Leverage Points. *Journal of the American Statistical Association*, 85 (411), 633–639. URL <http://www.jstor.org/stable/2289995>
- RStudio Inc. (2014a). ggviz: Interactive grammar of graphics [Software]. URL <http://CRAN.R-project.org/package=ggviz> (R package version 0.4)
- RStudio Inc. (2014b). rmarkdown: Dynamic documents for R [Software]. URL <http://rmarkdown.rstudio.com> (R package version 0.3.3)
- RStudio Inc. (2014c). RStudio [Software]. URL <http://www.rstudio.org/> (Version 0.98.1062)
- Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R*. New York, NY: Springer. URL <http://lmdvr.r-forge.r-project.org/>
- Shumway, R. H. & Stoffer, D. S. (2011). *Time Series Analysis and Its Applications* (3. Aufl.). New York, NY: Springer. URL <http://www.stat.pitt.edu/stoffer/tsa3/>
- Signorell, A. (2014). DescTools: Tools for descriptive statistics [Software]. URL <http://CRAN.R-project.org/package=DescTools> (R package version 0.99.8)
- Spector, P. (2008). *Data Manipulation with R*. New York, NY: Springer.
- Strang, G. (2003). *Lineare Algebra*. Berlin: Springer.
- Su, Y.-S., Gelman, A., Hill, J. & Yajima, M. (2011). Multiple imputation with diagnostics (mi) in R: Opening windows into the black box. *Journal of Statistical Software*, 45 (2), 1–67. URL <http://www.jstatsoft.org/v45/i02/>
- Templ, M. (2014). CRAN task view: Official statistics & survey methodology. URL <http://CRAN.R-project.org/view=OfficialStatistics> (Version 2014-08-18)
- Temple Lang, D., Swayne, D., Wickham, H. & Lawrence, M. (2014). rggobi: Interface between R and GGobi [Software]. URL <http://CRAN.R-project.org/package=rggobi> (R package version 2.1.20)
- Therneau, T. (2014). A Package for Survival Analysis in S [Software]. URL <http://CRAN.R-project.org/package=survival> (R package version 2.37-7)
- TIBCO Software Inc. (2013). TIBCO enterprise runtime for R (TERR) [Software]. URL <http://spotfire.tibco.com/> (Version 2.5)

- Tingley, D., Yamamoto, T., Keele, L. & Imai, K. (2014). mediation: R Package for Causal Mediation Analysis [Software]. URL <http://CRAN.R-project.org/package=mediation> (R package version 4.4.2)
- Urbanek, S. & Horner, J. (2014). Cairo: R graphics device using cairo graphics library for creating high-quality bitmap (PNG, JPEG, TIFF), vector (PDF, SVG, PostScript) and display (X11 and Win32) output [Software]. URL <http://CRAN.R-project.org/package=Cairo> (R package version 1.5-6)
- Vaidyanathan, R. (2013). rCharts: Interactive Charts using Javascript Visualization Libraries [Software]. URL <http://rcharts.io/> (R package version 0.4.5)
- van Buuren, S. (2012). *Flexible imputation of missing data*. Boca Raton, FL: Chapman & Hall/CRC.
- van Buuren, S. & Groothuis-Oudshoorn, K. (2011). MICE: Multivariate Imputation by Chained Equations in R. *Journal of Statistical Software*, 45 (3), 1–67. URL <http://www.jstatsoft.org/v45/i03/>
- Venables, W. N. & Ripley, B. D. (2002). *Modern Applied Statistics with S* (4. Aufl.). New York, NY: Springer. URL <http://www.stats.ox.ac.uk/pub/MASS4/>
- Venables, W. N., Smith, D. M. & the R Development Core Team. (2014). An introduction to R [Software-Handbuch]. Vienna, Austria. URL <http://CRAN.R-project.org/doc/manuals/R-intro.html>
- Verzani, J. (2014). *Using R for Introductory Statistics* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC.
- Wahlbrink, S. (2014). Eclipse Plug-In for R: StatET [Software]. URL <http://www.walware.de/goto/statet> (Version 3.4)
- West, B. T., Welch, K. B. & Galecki, A. T. (2014). *Linear Mixed Models: A Practical Guide Using Statistical Software* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC. URL <http://www-personal.umich.edu/~agalecki/>
- Wickham, H. (2007). Reshaping data with the reshape package. *Journal of Statistical Software*, 21 (12), 1–20. URL <http://www.jstatsoft.org/v21/i12/>
- Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis*. New York, NY: Springer. URL <http://ggplot2.org/book/>
- Wickham, H. (2012). stringr: Make it easier to work with strings [Software]. URL <http://CRAN.R-project.org/package=stringr> (R package version 0.6.2)
- Wickham, H. (2014a). *Advanced R programming*. Boca Raton, FL: Chapman & Hall/CRC. URL <http://adv-r.had.co.nz/>
- Wickham, H. (2014b). devtools: Tools to make developing R code easier [Software]. URL <http://CRAN.R-project.org/package=devtools> (R package version 1.6.1)
- Wickham, H. (2014c). *R packages*. URL <http://r-pkgs.had.co.nz/>
- Wickham, H., Cook, D., Hofmann, H. & Buja, A. (2011). tourr: An R package for Exploring Multivariate Data with Projections. *Journal of Statistical Software*, 40 (2), 1–18. URL <http://www.jstatsoft.org/v40/i02/>
- Wickham, H. & Francois, R. (2014). dplyr: A grammar of data manipulation [Software]. URL <http://CRAN.R-project.org/package=dplyr> (R package version 0.3.0.2)
- Wickham, H., James, D. A., Falcon, S., SQLite Authors, Healy, L. & RStudio Inc. (2014). RSQLite: SQLite interface for R [Software]. URL <http://CRAN.R-project.org/package=RSQLite> (R package version 1.0.0)
- Wilkinson, G. N. & Rogers, C. E. (1973). Symbolic description of factorial models for analysis of variance. *Applied Statistics*, 22, 392–399.

Literaturverzeichnis

- Wilkinson, L. (2005). *The Grammar of Graphics* (2. Aufl.). New York, NY: Springer.
- Wirtz, M. & Caspar, F. (2002). *Beurteilerübereinstimmung und Beurteilerreliabilität*. Göttingen: Hogrefe.
- Wood, S. N. (2006). *Generalized Additive Models: An Introduction with R*. Boca Raton, FL: Chapman & Hall/CRC.
- Würtz, D., Chalabi, Y. & Maechler, M. (2013). timeDate: Rmetrics – Chronological and Calendar Objects [Software]. URL <http://CRAN.R-project.org/package=timeDate> (R package version 3010.98)
- Xie, Y. (2013). *Dynamic Documents with R and knitr*. Boca Raton, FL: Chapman & Hall/CRC.
- Yan, J. & Fine, J. P. (2004). Estimating equations for association structures. *Statistics in Medicine*, 23 (6), 859–874.
- Yee, T. W. (2010). The VGAM Package for Categorical Data Analysis. *Journal of Statistical Software*, 32 (10), 1–34. URL <http://www.jstatsoft.org/v32/i10/>
- Zeileis, A. (2004). Econometric Computing with HC and HAC Covariance Matrix Estimators. *Journal of Statistical Software*, 11 (10), 1–17. URL <http://www.jstatsoft.org/v11/i10/>
- Zeileis, A. (2005). CRAN Task Views. *R News*, 5 (1), 39–40. URL <http://CRAN.R-project.org/doc/Rnews/>
- Zeileis, A. & Hothorn, T. (2002). Diagnostic Checking in Regression Relationships. *R News*, 2 (3), 7–10. URL <http://CRAN.R-project.org/doc/Rnews/>
- Zeileis, A., Kleiber, C. & Jackman, S. (2008). Regression Models for Count Data in R. *Journal of Statistical Software*, 27 (8), 1–25. URL <http://www.jstatsoft.org/v27/i08/>

Index

- (2×2)-Kontingenztafel, 366
- Abstand, 423
- AIC, 185, 291
- AICc, 185
- allgemeines lineares Modell, 462
- Codierschema, 465
 - Designmatrix, 172, 462
 - Hat-Matrix, 433, 474
 - Kontrastmatrix, 465
 - lineare Hypothesen, 475
 - Modellvergleiche, 475, 480
 - Parameterschätzungen, 473
 - parametrische Funktionen, 475
 - Teststatistiken, 482
- α -Adjustierung, 226, 228, 229, 252, 253, 277, 390
- α -Fehler, 171
- Anderson-Darling-Test, 357
- Anführungszeichen, 23, 106, 119
- anonyme Funktion, *siehe* Funktion
- anpassen, *siehe* Optionen
- Ansari-Bradley-Test, 383
- Anzahl, *siehe* Vektor
- Arbeitsverzeichnis, *siehe* Verzeichnis
- Architect, 4
- area under the curve (AUC), 369
- Arithmetik, 13
- array, 86
- Assoziativität, 14
- Attribut, 20
- Ausführungsreihenfolge, *siehe* Assoziativität
- äußeres Produkt, 75
- Auswahl, *siehe* Daten
- AV, 172
- Balkendiagramm, *siehe* Grafik
- Batch-Modus, 153
- beenden, 10
- Befehlshistorie, 12
- Befehlsskript, *siehe* Skript
- Beobachterübereinstimmung, *siehe* Inter-Rater-Übereinstimmung
- β -Fehler, 171
- BIC, 185
- Binomialtest, 352, 386
- power, 278
- Binomialverteilung, *siehe* Verteilung
- Bootstrap-Verfahren, *siehe* Resampling-Verfahren
- Bowker-Test, 395
- Box-Cox-Transformation, *siehe* Regression
- Boxplot, *siehe* Grafik
- Bradley-Terry-Modelle, 305
- Brier-Score, 495
- Brown-Forsythe-Test, 214
- CART-Modelle, 458
- Chancenverhältnis, 368
- χ^2 -Test
- Auftretenswahrscheinlichkeiten, 363
 - feste Verteilung, 358
 - Gleichheit von Verteilungen, 362
 - Normalverteilung, 359
 - Unabhängigkeit, 361
 - Verteilungsklasse, 359
- χ^2 -Verteilung, *siehe* Verteilung
- Cholesky-Zerlegung, *siehe* Matrix
- Clusteranalyse, 458
- Cochran-Mantel-Hänszel-Test, 365
- Cochran- Q -Test, 394
- Cohens d , 216, 218, 219
- Cohens κ , *siehe* Inter-Rater-Übereinstimmung
- connection, 157
- Cox proportional hazards Modell, *siehe* Survival-Analse
- Cramérs V , 373
- CRAN, 16
- d , *siehe* Cohens d
- Dateien
- auflisten, 167

INDEX

- manipulieren, 169
Pfad, Name, Endung, 168
- Daten
Ausreißer, 196, 534, 545
auswählen, 47
Dateneditor, 155
diskretisieren, 53, 62
doppelte Werte, 37, 135
eingeben, 154
ersetzen, 51
Extremwerte, 196
fehlende Werte, 14, **96**, 102, 135
ausschließen, 99
codieren, 97
ersetzen, 103
fallweiser Ausschluss, 100
identifizieren, 97
Indexvektor, 36
multiple Imputation, 103
paarweiser Ausschluss, 101
recodieren, 98
sortieren, 102
zählen, 98
- Fishers Z-Transformation, 180
getrennt nach Gruppen auswerten, 73
- Import / Export, 154
R-Format, 158
andere Programme, 158
Datenbank, 161
Konsole, 165
online, 157
Textformat, 155, 157, 158
Zwischenablage, 156, 157
- interpolieren, 547
Kennwerte, 63
Median-Split, 62
Qualität, i, 96, 135
Rangtransformation, 50
recodieren, 51, 58
skalieren, 50
speichern, 154
Viewer, 124
wiederholen, 43
winsorisieren, 67
 z -transformieren, 49
zentrieren, 49
Zufallsauswahl, 47
- Datenbank, *siehe* Daten
Datensatz, 122
Aufbau, 123, 141, 143
Dimensionierung, 123
Dokumentation, 18
doppelte Werte, 135
Elemente auswählen und verändern, 125
erstellen, 122
fehlende Werte, 135
Funktionen anwenden, 147, 150, 151
indizieren, 125
Long-Format, 142, **143**, 232, 254, 261
Pakete, 18, 122
sortieren, 131
Suchpfad, 128
teilen, 136
Teilmengen auswählen, 132
umformen, 141, 143
Umwandlung in Datensatz, 125
Variablen hinzufügen, 130
Variablen löschen, 131
Variablennamen, 127
verbinden, 137
Wide-Format, 142, **143**, 232, 259
- Datentyp, 20, 23
mischen, 31, 76, 117, 122
umwandeln, 23
- deskriptive Kennwerte, *siehe* Daten
Determinante, *siehe* Matrix
Determinationskoeffizient, *siehe* Regression
Devianz, *siehe* Regression
Dezimaltrennzeichen, 160
Diagramm, *siehe* Grafik
Differenzen, 63
Dimensionierung, *siehe* Datensatz, Matrix
Diskriminanzanalyse, 457
Distanzmaß, *siehe* Abstand
Diversitätsindex, 70
Dokumentation, *siehe* Literatur
download, 3, 5
Dummy-Codierung, *siehe* Varianzanalyse
- eager evaluation, 578
Eclipse, 4
Editor, 4, 153
Effektcodierung, *siehe* Varianzanalyse
Effektstärke, *siehe* t-Test, Varianzanalyse

INDEX

- Effizienz, 587
Eigenwerte, -vektoren, *siehe* Matrix
Eingabe, *siehe* Daten
Einstellungen, 10
ENTWEDER-ODER, 24
environment, *siehe* Umgebung
Ereigniszzeiten, *siehe* Survival-Analse
Escape-Sequenz, 23, 106
 $\hat{\eta}^2$ (Effektstärke), 223, 235, 241, 257, 262, 275
Excel, 159, 161
Exponentialfunktion, 14
Extremwerte, 64
 f (Effektstärke), 223
 F -Maß, *siehe* (2×2)-Kontingenztafel
 F -Verteilung, *siehe* Verteilung
Faktor, 54
Faktorenanalyse, 439
Fakultät, 64
Fallunterscheidung, 571
Fallzahlberechnung, 278
FALSCH, 24
Faltung, 534
FAQ, 5
Farben, *siehe* Grafik
fehlende Werte, *siehe* Daten
Fehlerbalken, *siehe* Grafik
Fehlermeldung, 19
Fishers exakter Test
 Gleichheit von Verteilungen, 365
 Unabhängigkeit, 364
Fleiss' κ , *siehe* Inter-Rater-Übereinstimmung
Fligner-Killeen-Test, 214
format string, 105, 112
Fourier-Transformation, 534
Friedman-Test, 392
Funktion
 anonyme, 582
 Argumente, 15, 577
 aufrufen, 15, 581
 debuggen, 584
 erstellen, 577
 generische, 582
 Geschwindigkeit, 587
 getrennt nach Gruppen anwenden, 73
 Kopf, 577
 Methode, 582
 polymorphe, 582
 profilen, 587
 Quelltext, 584
 Rumpf, 578
 Rückgabewert, 581
 unbenannte, 582
 überladene, 582
 Funktionsgraph, *siehe* Grafik
 Γ -Funktion, 64
GEE (generalized estimating equations), 208
gemischte Modelle, 208
Genauigkeit, *siehe* Zahlen
generische Funktion, *siehe* Funktion
geordnete Paare, 75
Geschwindigkeit, *siehe* Effizienz
Gleichverteilung, *siehe* Verteilung
Globbing-Muster, *siehe* Zeichenketten
Goodman und Kruskals γ , 373
Grafik
 3D, 549
 3D Gitter-Diagramm, 552
 3D Streudiagramm, 553
 Achsen, 511, 528
 Ausgabe, 499
 Balkendiagramm, 511
 Basissystem, 499
 bedingte Diagramme, 553
 Beschriftung, 526–528
 Bildbearbeitung, 534
 boxplot, 537
 clipping, 507, 518
 Datenpunkte identifizieren, 504
 Datenpunktssymbole, 507
 device, 499
 unterteilen, 565
 verwalten, 499–501
 dotchart, 515
 Einstellungen, 505
 Elemente hinzufügen, 516
 Farben, 508
 Fehlerbalken, 529
 Fenster öffnen, 499
 Formate, 501
 formatieren, 505
 Funktionsgraph, 525

INDEX

- gemeinsame Verteilung, 543
Gerade, 520
Gitter, 520
Glätter, 547
heatmap, 563
High-Level-Funktion, 499, 499, 516, 567
Histogramm, 534
Höhenlinien, 550, 551
interaktive, 499, 553
interpolieren, 548
konvexe Hülle, 545
Koordinaten identifizieren, 517
Koordinatensysteme, 518
kopieren, 502
Kreisdiagramm, 542
kumulierte Häufigkeiten, 542
Legende, 526
Linien, 520
Liniendiagramm, 502, 505
Liniensegmente, 521
Linientypen, 507
Low-Level-Funktion, 499, 516
mathematische Formeln, 528
mehrere Grafiken in einem device, 565
multivariate Daten, 549
Optionen, 505
Pfeile, 521
Polygone, 523
Punktdiagramm, 515
Punkte, 519
Quantil-Quantil-Diagramm, 540
Rahmen, 522
Rastergrafik, 532
Rechtecke, 522
Regionen, 499
Ränder, 499
Säulendiagramm, 511
Scree-Plot, 439
Seitenverhältnis, 520
speichern, 501
Stamm-Blatt-Diagramm, 536
Streudiagramm, 502, 505
Streudiagramm-Matrix, 561
Streuungsellipse, 543
stripchart, 539
Symbole, 528
Säulendiagramm, 512
Text, 527
Titel, 526
Transparenz, 510
grafische Benutzeroberfläche, 4
Groß- und Kleinschreibung, 20
GUI, *siehe* grafische Benutzeroberfläche
 H_0, H_1 , 171
Häufigkeiten
 absolute, 88
 bedingte relative, 91
 Iterationen, 90
 Kreuztabelle, 90
 kumulierte relative, 89, 95
 Randkennwerte, 94
 relative, 89
Hauptkomponentenanalyse, 433
herunterladen, *siehe* download, *siehe* download
Hilfe
 eingebautes Hilfesystem, 16
 Internet, 5
 Literatur, 5
 Zusatzpakete, 18
Histogramm, *siehe* Grafik
Hodges-Lehmann-Schätzer, 67, 388, 390
Homepage, 3, 5
Hotelling-Lawley-Spur, 456, 482
Hotellings T^2
 eine Stichprobe, 449
 zwei abhängige Stichproben, 453
 zwei unabhängige Stichproben, 451
Huber-M-Schätzer, 68
ICC, *siehe* Inter-Rater-Übereinstimmung
Indexvektor, *siehe* Vektor
indizieren, *siehe* Datensatz, Matrix, Vektor
Installation, 3
Inter-Rater-Übereinstimmung, 374
 r_{WG} -Koeffizienten, 374
 Cohens κ , 375, 377
 Fleiss' κ , 376
 Intra-Klassen-Korrelation, 380
 Kendalls W , 379
 prozentuale Übereinstimmung, 375
Interquartilabstand, 72

INDEX

- Intra-Klassen-Korrelation, *siehe* Inter-Rater-Übereinstimmung
Iterationen, *siehe* Häufigkeiten
Iterationslängentest, *siehe* Runs-Test
Jonckheere-Terpstra-Test, 392
Kendalls τ , 371
Kendalls W , *siehe* Inter-Rater-Übereinstimmung
Kern, *siehe* Matrix
Klammern, 14
Klasse, 19
Klassifikationsverfahren, 293, 458
Kolmogorov-Smirnov-Test
 Anpassungstest, 355
 Gleichheit von Verteilungen, 384
 Lilliefors-Schranken, 357
 Normalverteiltheit, 355
Kombinatorik, 39
Kommentar, 8, 18, 154, 572
komplexe Zahlen, *siehe* Zahlen
Kondition κ , *siehe* Matrix
Konfidenzintervall, 172
Konfusionsmatrix, 366
Konsole, 6
Konstanten, 13
Kontingenzkoeffizient, 373
Kontingenztafel, *siehe* Häufigkeiten
Kontraste, *siehe* Varianzanalyse
Kontrollstrukturen, 571
konvexe Hülle, *siehe* Grafik
Korrelation, 71
 kanonische, 71
 Kendalls τ , 371
 Korrelationsmatrix, 85
 multiple Korrelation, 184
 Partialkorrelation, 71, 208
 polychorische, 71
 polyseriale, 71
 Rangkorrelation, 371
 Semipartialkorrelation, 71, 209
 Spearmans ρ , 371
 Test, 179
Kovarianz, 70
 Kovarianzmatrix, 85
 Rangkovarianz, 371
 robuste Schätzer, 73
 unkorrelierte, 70, 85
Kovarianzanalyse, 270
 Effektstärke, 275
 Einzelvergleiche (Kontraste), 276, 277
Kreuztabelle, *siehe* Häufigkeiten
Kreuzvalidierung, 492
 k-fache, 493
 Kreuzvalidierungsfehler, 493
 leave-one-out, 494
 verallgemeinerte, 495
Kruskal-Wallis- H -Test, 390
kumulierte relative Häufigkeiten, *siehe* Häufigkeiten
kumulierte Summe, *siehe* Summe
kumuliertes Produkt, *siehe* Produkt
Kurtosis, *siehe* Wölbung
Länge
 Norm, 422
 Vektor, 20
 Zeichenkette, 104
lazy evaluation, 578
leere Menge, 14
Levene-Test, 213
Lilliefors-Test, *siehe* Kolmogorov-Smirnov-Test
lineare Algebra, *siehe* Matrix
lineare Gleichungssysteme lösen, 422
lineare Strukturgleichungsmodelle, 439
Liste, 117
Literatur, 5
logische Operatoren, *siehe* Vergleiche
logische Vergleiche, *siehe* Vergleiche
logische Werte, *siehe* Wahrheitswerte
logistische Regression, *siehe* Regression
Logit-Funktion, 287
Long-Format, *siehe* Datensatz
Mahalanobisdistanz, 425
Mahalanobistransformation, 424
Mailing-Liste, 5
Mann-Whitney- U -Test, 390
Mantel-Hänszel-Test, *siehe* Survival-Analyse
Matrix
 Addition, 419
 Algebra, 419
 Cholesky-Zerlegung, 430

INDEX

- Datentypen, 76
Determinante, 426
Diagonalelemente, 419
diagonalisieren, 428
Diagonalmatrix, 419
Dimensionierung, 76
Dreiecksmatrix, 78
Eigenwerte, -vektoren, 427, 433
Einheitsmatrix, 419
Elemente auswählen und verändern, 78,
 80
erstellen, 75, 81
Funktionen anwenden, 83
Hadamard-Produkt, 419
indizieren, 78, 80
Inverse, 422
Kern, 422
Kondition κ , 202, 428
Kronecker-Produkt, 419
mit Kennwert verrechnen, 84
Multiplikation, 419
Norm, 423
orthogonale Projektion, 430
Pseudoinverse, 187, 422, 433
QR-Zerlegung, 427, 430
Randkennwerte, 83
Rang, 427
Singulärwertzerlegung, 429, 439
Skalarprodukt, 419
sortieren, 82
Spaltennorm, 422
Spektralzerlegung, 428
Spur, 426
SSP-Matrix, 420
transponieren, 418
umwandeln, 77
Umwandlung in Matrix, 77
verbinden, 81
Wurzel, 429
Zeilen- und Spaltenindex, 77
zentrieren, 84, 420
Mauchly-Test, 237, 238
Maximum, 64
McNemar-Test, 396
Median, 65
Mediationsanalyse, *siehe* Regression
Mengen, 37
Methode, *siehe* Funktion
Minimum, 64
Mittelwert, 65
 geometrischer, 65
 gestutzter, 67
 harmonischer, 65
 winsorisierte, 67
Mittelwertstabelle, 74
Modalwert, 66
Modellformel, 172
Modus, *siehe* Datentyp
Mood-Test, 382
multidimensionale Skalierung, 446
Multinomiale Regression, *siehe* Regression
multiple Imputation, *siehe* Daten
multivariate Verfahren, 418
multivariate z -Transformation, 424
nonparametrische Methoden, 351
Norm, 422
Normalengleichungen, 430
Normalverteilung, *siehe* Verteilung
Notation, 3
Objekt, 19
 Größe, 587
 Gültigkeit, 578
 label, 20
 Lebensdauer, 578
 lokales, 578
 löschen, 22
 Maskierung, 20
 Namenskonflikte, 20
 Objektnamen, 20, 22
 verstecktes, 22
Objektorientierung, 582
ODBC, 161
odds ratio, 289, 368
ODER, 24, 33, 34
 $\hat{\omega}^2$ (Effektstärke), 223
OpenOffice, 159
Operatoren, 13
Optimierung, *siehe* Effizienz
Optionen, 11
Ordinale Regression, *siehe* Regression
Ordner, *siehe* Verzeichnis
Page-Trend-Test, 393

INDEX

- Pakete, 16
Partialkorrelation, *siehe* Korrelation
Permutation, *siehe* Kombinatorik
Permutationstest, *siehe* Resampling-Verfahren
 ϕ -Koeffizient, 373
Pillai-Bartlett-Spur, 455, 482
Poisson-Regression, *siehe* Regression
Polymorphie, *siehe* Funktion
positiver Vorhersagewert, *siehe* (2×2) -Kontingenztafel
power, 172, 278
Probit-Regression, *siehe* Regression
Produkt, 64
programmieren, 571
Projektion, *siehe* Matrix
Prompt, 7
Protokoll, 7
Prozentrang, 68, 95
Prävalenz, *siehe* (2×2) -Kontingenztafel
Präzision, *siehe* (2×2) -Kontingenztafel
Pseudoinverse, *siehe* Matrix
- Q -Koeffizient, *siehe* Yules Q -Koeffizient
 Q -Test, *siehe* Cochran- Q -Test
 QR -Zerlegung, *siehe* Matrix
Quadratsummen-Typen, *siehe* Varianzanalyse
Quantil, 68
Quartil, 68
- r_{WG} -Koeffizienten, 374
Rang, *siehe* Matrix
range, 65
Rangkorrelation, *siehe* Korrelation
Rangplatz, *siehe* Daten
Rate der korrekten Klassifikation, *siehe* (2×2) -Kontingenztafel
recall, *siehe* (2×2) -Kontingenztafel
recodieren, *siehe* Daten
recycling, *siehe* Vektor
Regression
 bedingte logistische, 288
 Box-Cox-Transformation, 200
 Determinationskoeffizient R^2 , 184
 Devianz, 291
 Einfluss, 197
 elastic net, 205
 grafische Darstellung, 182, 187
- Hat-Matrix, 187, 197
Hauck-Donner-Phänomen, 295
Hebelwert, 197
hierarchische, 189
Homoskedastizität, 200, 473
Hurdle-Regression, 317
Interaktion, 191
Konfidenzintervall, 185, 194
Kreuzvalidierung, *siehe* Kreuzvalidierung
LASSO, 205
lineare, 180, 186
logistische, 288
Mediation, 183
Modell verändern, 188
Modellvergleich, 189
Moderation, 191
Multikollinearität, 201
multinomiale, 304
multivariate, 447, 462, 483
negative Binomial-Regression, 312
nichtlineare, 208
ordinale, 298
penalisierte, 204
Poisson-Regression, 309
polynomiale, 207
PRESS-Residuen, 494
Probit-Regression, 297
psueduo- R^2 , 291
Regressionsanalyse, 183, 483
Regressionsdiagnostik, 195
Residuen, 182, 291
Ridge-Regression, 205
robuste, 203
Sobel-Test, 183
standardisierte, 182
Stepwise-Verfahren, 189
Toleranzintervall, 194
Varianzinflationsfaktor, 202
Voraussetzungen, 195
zero-inflated Modelle, 314
zero-truncated Modelle, 317
regulärer Ausdruck, *siehe* Zeichenketten
Relevanz, *siehe* (2×2) -Kontingenztafel
Resampling-V erfahren, 400
 Bootstrapping, 400
 Konfidenzintervalle, 406
 stratifiziert, 406

INDEX

- Bootstrapping
case resampling, 407
model-based resampling, 409
wild resampling, 411
Konfidenzintervalle, 404
Permutationstests, 412
robuste Verfahren
Ausreißer identifizieren, 196
Diskriminanzanalyse, 458
Hauptkomponentenanalyse, 434
Kovarianzschätzer, 73
Regression, 203
Streuungsmaße, 72
zentrale Tendenz, 67
ROC-Kurve, 369
Roys Maximalwurzel, 456, 482
RStudio, 4, 6
Runs-Test, 353
Rückgabewert, *siehe* Funktion

S, 1
S3-, S4-Paradigma, 582
SAS, 161
Säulendiagramm, *siehe* Grafik
Schiefe, 69
Schleifen, 574
Schlüsselwörter, 20
Scoping-Regeln, *siehe* Objekte
Scree-Plot, *siehe* Grafik
Semipartialkorrelation, *siehe* Korrelation
Sensitivität, *siehe* (2×2)-Kontingenztafel
Shannon-Index, 70
Shapiro-Wilk-Test, 357
Sign-Test, *siehe* Vorzeichen-Test
Signalverarbeitung, 534
Simulation, 123, 574
Singulärwertzerlegung, *siehe* Matrix
Skalarprodukt, *siehe* Matrix
Skript, 153
Sobel-Test, *siehe* Regression
Somers' d , 373
sortieren, *siehe* Datensatz, Matrix, Vektor
Spannweite, *siehe* range
Spearmans ρ , 371
speichern, *siehe* workspace
Spektralzerlegung, *siehe* Matrix
Spezifität, *siehe* (2×2)-Kontingenztafel

splines, 207, 548
SPSS, 159, 243, 298
starten, 6
Stata, 161
Stichprobengröße, 278
Streudiagramm, *siehe* Grafik
Streuung, 69
Streuungsellipse, *siehe* Grafik
Stuart-Maxwell-Test, 398
Stufenfunktion, 95
Suchpfad, 12, *siehe* Datensatz
Summe, 63
Summenscore, 51
support vector machines, 458
Survival-Analyse, 324
Cox proportional hazards Modell, 334
Kaplan-Meier, 331
Log-Rank-Test, 333
Mantel-Hänszel-Test, 333
parametrische Modelle, 345
wiederkehrende Ereignisse, 329
zeitabhängige Prädiktoren, 330
Zählprozess-Darstellung, 328

t-Test
Effektstärke, 216, 218, 219
eine Stichprobe, 215
power, 280, 281
Stichprobengröße, 281
zwei abhängige Stichproben, 219
zwei unabhängige Stichproben, 217

t-Verteilung, *siehe* Verteilung
Tabellenkalkulation, 159
Task Views, 16
Tastaturkürzel, 10, 153
Teststärke, *siehe* power
Text, *siehe* Grafik, Zeichenkette
Tjur Diskriminationsindex, 292
transponieren, *siehe* Matrix
Treatment-Kontraste, *siehe* Varianzanalyse
trigonometrische Funktionen, 14
Tschuprows T , 373

Uhrzeit, 113
Umgebung, 11, 578
unbenannte Funktion, *siehe* Funktion

INDEX

- UND, 24, 33
UV, 172
- Variable
kategoriale, *siehe* Faktor
latente, 439
neue aus bestehenden bilden, 51
- Varianz, 69
unkorrigierte, 69, 85
winsorisierte, 72
- Varianzanalyse
assoziierte einfaktorielle, 249, 251
bedingte Haupteffekte, 246
Blockeffekt, 232, 254
dreifaktorielle
abzählige Gruppen (RBF- pqr), 255
Split-Plot (SPF- $p \cdot qr$), 267
Split-Plot (SPF- $pq \cdot r$), 269
unabhängige Gruppen (CRF- pqr), 240
- Dummy-Codierung, 245, 466
Effektcodierung, 245, 467
Effektstärke, 223, 235, 241, 257, 262
einfaktorielle
abzählige Gruppen (RB- p), 231, 237, 454
unabhängige Gruppen (CR- p), 220, 221, 223
fester Effekt, 232
Helmert-Codierung, 467
- Kontraste
a-priori, 225, 249
paarweise t -Tests, 229
post-hoc (Scheffé), 228, 252
Tukey HSD, 230, 253
zweifaktoriell, 252
- multivariate
einfaktorielle, 455, 464, 476, 485
zweifaktorielle, 456, 469, 476, 488
- orthogonales Design, 239, 242
power, 283
Quadratsummen-Typen, 190, 242, 476
Random-Faktor, 232, 254, 261
simple effects, 246
Stichprobengröße, 283
Treatment-Kontraste, 466
Voraussetzungen, 224
Zirkularität, 232, 235, 258, 264
- ε -Korrektur, 235, 237, 238, 258, 264
Mauchly-Test, 237, 238
- zweifaktorielle
abzählige Gruppen (RBF- pq), 254
Split-Plot (SPF- $p \cdot q$), 261
unabhängige Gruppen (CRF- pq), 239
- Varianzhomogenität
 F -Test, 212
Ansari-Bradley-Test, 383
Brown-Forsythe-Test, 214
Fligner-Killeen-Test, 214
Levene-Test, 213
Mood-Test, 382
- Vektor, 20, 27
Anzahl Elemente, 28
Bedingungen prüfen, 33
Datentypen, 31
Elemente auslassen, 29
Elemente auswählen, 28
Elemente benennen, 31
Elemente ersetzen, 51
Elemente löschen, 32
Elemente ändern, 30
elementweise Berechnung, 48
erstellen, 27
Fälle zählen, 35
- Indexvektor
logisch, 35
numerisch, 29, 36
indizieren, 28
- Kreuzprodukt, 419
Länge, 28
Norm, 422
recycling, 49
Reihenfolge, 45, 46
- Skalarprodukt, *siehe* Matrix
sortieren, 46
unbenannt, 29
Vergleiche, 33
verkürzen, 32
verlängern, 30
zusammenfügen, 28
zyklische Verlängerung, 49
- Verallgemeinerte Schätzgleichungen, 208
verallgemeinertes lineares Modell, 287
Vergleiche, 24
- Verteilung

INDEX

- F*-Verteilung, 44, 175
 χ^2 -Verteilung, 44, 175
t-Verteilung, 44, 175
Binomialverteilung, 44, 175
Dichtefunktion, 175
Dichtefunktion schätzen, 536
Gleichverteilung, 44, 175
Hypergeometrische Verteilung, 175
kritischer Wert, 171, 177
Nonzentralitätsparameter, 280, 282, 284
Normalverteilung, 44, 175
 χ^2 -Test, 359
Anderson-Darling-Test, 357
Kolmogorov-Smirnov-Test, 355
Lilliefors-Test, 357
Shapiro-Wilk-Test, 357
p-Wert, 171, 177
Poisson-Verteilung, 175
Quantilfunktion, 177
Verteilungsfunktion, 176
Wahrscheinlichkeitsfunktion, 175
Wilcoxon-Vorzeichen-Rang-Verteilung, 175
verteilungsfreie Methoden, *siehe* nonparametrische Methoden
Vertrauensintervall, *siehe* Konfidenzintervall
Verzeichnis
 Arbeitsverzeichnis, 10
 Pfadangabe, 167
Verzweigung, 571
Vorzeichen-Test, 386
Vuong-Test, 316

WAHR, 24
Wahrheitswerte, 23, 24
Wald-Wolfowitz-Test, 355
Wide-Format, *siehe* Datensatz
Wilcoxon-Test
 abhängige Stichproben, 390
 Rangsummen-Test, 389
 Vorzeichen-Rang-Test, 387
Wilcoxon-Vorzeichen-Rang-Verteilung, *siehe*
 Verteilung
Wilks' Λ , 456, 482
Winkelfunktionen, *siehe* trigonometrische Funktionen
Wölbung, 69
workspace, 11

Yules *Q*-Koeffizient, 368

Zahlen
 Betrag, 13
 Dezimalstellen, 13
 Dezimalteil, 13
 Dezimaltrennzeichen, 13, 159
 e, 14
 Exponentialschreibweise, 13
 Fakultät, 14
 ganze, 23
 Genauigkeit, 26
 Gleitkommazahlen, 23, 26
 Grundrechenarten, 13
 i, 14
 komplexe, 14, 23
 Logarithmus, 14
 π , 14
 Quadratwurzel, 13
 reelle, 23
 runden, 13
 unendlich, 14
 Vorzeichen, 13
 Zahlenfolgen erstellen, 42
Zeichenketten, 23
ausführen, 111
Globbing-Muster, 110
Groß- / Kleinbuchstaben, 107
Länge, 104
nach Muster erstellen, 104
Platzhalter, 110
regulärer Ausdruck, 109
umkehren, 108
Umwandlung in Faktor, 124, 156
Umwandlung in Zeichenketten, 103
verbinden, 106
wildcards, 110
Zeichenfolgen ersetzen, 110
Zeichenfolgen extrahieren, 109
Zeichenfolgen finden, 108
Zeilenumbruch, 106
zerlegen, 107
Zeit, *siehe* Uhrzeit
Zirkularität, *siehe* Varianzanalyse
Zufallsauswahl, *siehe* Daten
Zufallseffekt, *siehe* Varianzanalyse
Zufallsvariablen, 174

INDEX

- Zufallszahlen, 41, 43, 44
 - zufällige Reihenfolge, 46
 - Zusammenhangsmaße, 373
 - ϕ -Koeffizient, 373
 - r_{WG} -Koeffizienten, 374
 - Cramérs V , 373
 - Goodman und Kruskals γ , 373
 - Kontingenzkoeffizient, 373
 - Korrelation, 71
 - Rangkorrelation, 371
 - Somers' d , 373
 - Tschuprows T , 373
 - Zusatzpakete, *siehe* Pakete
 - Zuweisung, 21
 - Zwischenablage, 156, 157, 159
 - zyklische Verlängerung, *siehe* Vektor
- Überlebenszeiten, *siehe* Survival-Analise

R-Funktionen, Klassen und Schlüsselwörter

!, 24
!=, 24
", ', 23, 106
\$, 119, 125
&, &&, 24, 33
((), 21, 577
*, 13, 173
+, 10, 13, 173
-, 13, 173
. ., 22, 173
. . ., 578
.First(), 11, 577
.GlobalEnv, 12
.Last(), 11
.Last.value, 22, 155
.Machine, 23, 27
.Platform, 7
.RData, 12
.Rhistory, 12
.Rprofile, 11
.libPaths(), 17
/, 13, 173
::, 42, 173
:::, 18, 584
::::, 584
;;, 8
<, 24
<-, 21
<=, 24
=, 21
==, 24
>, 7, 24
>=, 24
? , 16
[[]], 118, 125
[] , 28, 78, 87, 126
, 8, 18, 154
% , 105
%*%, 419
%/%, 13
%%, 13
%^%, 419

approx(), 547
 approfun(), 547
 apropos(), 16
 aq.plot(), 196
 Arg(), 14
 args(), 15
 array(), 86
 arrayInd(), 81
 arrows(), 521
 as.<Datentyp>(), 23
 as.<Klasse>(), 20
 as.data.frame(), 125
 as.Date(), 112
 as.list(), 125
 as.matrix(), 125
 as.POSIXct(), 114
 as.POSIXlt(), 114
 as.raster(), 532
 asin(), 14
 assign(), 22
 Assocs(), 373
 atan(), 14
 atan2(), 14
 attach(), 129
 attr(), 20
 attributes(), 20
 ave(), 73
 axis(), 529
 axis.break(), 529
 barchart(), 560
 barp(), 511
 barplot(), 511
 basehaz(), 339
 basename(), 168
 bcPower(), 201
 binom.test(), 352
 BinomCI(), 352
 biplot(), 439
 boot(), 401
 boot.ci(), 404
 boot.array(), 403
 box(), 522
 boxplot(), 538
 bptest(), 200
 brat(), 305
 break, 576
 browser(), 587
 bs(), 207
 bwplot(), 560
 by(), 151
 C(), 468
 c(), 27, 56
 Cairo(), 501
 cancor(), 71
 capture.output(), 103, 584
 cat(), 106
 cbind(), 81, 130, 138
 cdplot(), 288, 514
 ceiling(), 13
 character, 23
 chisq.test(), 358, 361
 chol(), 430
 choose(), 39
 choose.files(), 167
 chull(), 545
 class(), 19
 clogit(), 288
 close.screen(), 570
 cmdscale(), 446
 cmpfun(), 589
 cnvrt.coords(), 518
 coef(), 182, 201, 290, 473
 coeftest(), 204, 313
 CohenKappa(), 376, 377
 col(), 77
 col2rgb(), 509
 colMeans(), 83
 colnames(), 127
 colorRamp(), 510
 colorRampPalette(), 510
 colors(), 509
 colSums(), 83
 combn(), 39
 comment(), 20
 complete.cases(), 135
 complex, 23
 confidenceEllipse(), 529, 544
 confint(), 185, 226, 290, 301
 conflicts(), 20
 Conj(), 14
 contour(), 550
 contr.sum(), 468

```

contr.treatment(), 466
contrasts(), 469
convertColor(), 510
convolve(), 534
cor(), 71, 85, 371
cor.test(), 372
cos(), 14
cov(), 70, 85, 371
cov.wt(), 85
cov2cor(), 85, 420
covMcd(), 73
covOGK(), 73
cox.zph(), 341
coxnet(), 345
coxph(), 335
cross(), 419
crossprod(), 419
cumprod(), 64
cumsum(), 63
curve(), 525
cut(), 62
CutQ(), 63
cv.glm(), 493
cv.glmnet(), 205

data(), 18
data.frame(), 122, 138
data.matrix(), 125
Date, 112
date(), 113
db<Befehl>(), 163
dbinom(), 175
dchisq(), 175
debug(), 586
delayedAssign(), 578
demo(), 499, 553
density(), 536
deparse(), 112
Desc(), 63
det(), 426
detach(), 18, 129
dev.copy(), 502
dev.cur(), 500
dev.list(), 500
dev.new(), 499
dev.next(), 500
dev.off(), 501

dev.prev(), 500
dev.set(), 500
deviance(), 291
df(), 175
diag(), 419
diff(), 63
difftime(), 115
dim(), 76, 123
dimnames(), 127
dir(), 153
dir.create(), 169
dir.exists(), 169
dirname(), 168
dist(), 423
dmvnorm(), 551
dnorm(), 175
do.call(), 149
dotchart(), 515
dotplot(), 560
double, 23
dperm(), 414
drop, 79, 126
drop1(), 189, 245, 295
droplevels(), 58
dt(), 175
dummy.coef(), 473
dump(), 158
duplicated(), 37, 135
durbinWatsonTest(), 200

ecdf(), 95, 542
edit(), 155
effect(), 276
eigen(), 427
else, 572
Entropy(), 70
erase.screen(), 570
ErrBars(), 530
Error(), 233, 254, 261
EtaSq(), 223, 235, 242, 257, 263
eval(), 112
example(), 16
exists(), 20
exp(), 14
expand.grid(), 40, 61
expression(), 528
extractAIC(), 185

```

F, siehe FALSE
 fa.parallel(), 443
 facet_grid(), 557
 facet_wrap(), 557
 factanal(), 441
 factor(), 54
 factorial(), 14, 64
 FALSE, 24
 fft(), 534
 file.choose(), 167
 file.copy(), 169
 file.create(), 169
 file.exists(), 169
 file.remove(), 169
 file.rename(), 169
 file_ext(), 169
 file_path_sans_ext(), 169
 filled.contour(), 551
 findFn(), 16
 fisher.test(), 364, 366
 FisherZ(), 180
 FisherZInv(), 180
 fitdistr(), 547
 fitted(), 182, 292, 293
 fix(), 155
 fligner.test(), 214
 floor(), 13
 for(), 574
 format(), 113, 114
 formatC(), 103
 formula, 172
 fread(), 158, 588
 friedman.test(), 392
 ftable(), 92
 function(), 577
 gam(), 208
 gamma(), 64
 geom_bar(), 556, 557
 geom_boxplot(), 556, 558
 geom_histogram(), 556, 558
 geom_hline(), 559
 geom_line(), 556
 geom_point(), 555, 556
 geom_smooth(), 559
 geom_text(), 559
 geom_vline(), 559
 get(), 21, 584
 getAnywhere(), 584
 getOption(), 11
 getS3method(), 584
 getwd(), 10
 ggplot(), 554
 ggsave(), 555
 ggtile(), 556
 GiniMd(), 73
 ginv(), 422
 gl(), 61
 glht(), 225, 249, 266, 276
 glm(), 288, 310
 glm.nb(), 313
 glmnet(), 205
 glob2rx(), 110, 167
 Gmean(), 65
 GoodmanKruskalGamma(), 373
 graphics.off(), 501
 gray(), 510
 gregexpr(), 110
 grep(), 108
 grep1(), 109
 grid(), 520
 gsub(), 111, 160
 guides(), 556
 hatvalues(), 197
 hcl(), 510
 head(), 124
 heatmap(), 563
 help(), 16
 help.search(), 16
 help.start(), 16
 hexbin(), 546
 hist(), 535
 hist3d(), 553
 histbackback(), 535
 histogram(), 560
 history(), 12
 Hmean(), 65
 HodgesLehmann(), 67
 HotellingsT2Test(), 449, 451
 hsv(), 510
 huberM(), 68
 hurdle(), 317

I(), 123, 174
 ICC(), 380
 identical(), 25
 identify(), 504
 identity(), 74
 if(), 571
 ifelse(), 52
 Im(), 14
 Inf, 14
 influence.measures(), 198
 influenceIndexPlot(), 198
 influencePlot(), 198
 inherits(), 580
 install.packages(), 17
 install_github(), 17
 installed.packages(), 17
 integer, 23
 interaction(), 57, 240
 interaction.plot(), 241
 intersect(), 38
 inverse.rle(), 90
 invisible(), 581
 IQR(), 72
 is.Datentyp(), 23
 is.Klasse(), 19
 is.Speicherart(), 23
 is.element(), 38, 134
 is.finite(), 14
 is.infinite(), 14
 is.na(), 14, 97
 is.nan(), 14
 is.null(), 14
 ISOdate(), 114
 isTRUE(), 25, 34
 jitter(), 535, 545
 JonckheereTerpstraTest(), 392
 jpeg(), 501
 kappa(), 203
 Kappam(), 376
 KendallW(), 379
 knots(), 96
 kronecker(), 419
 kruskal.test(), 391
 ks.test(), 356, 384
 Kurt(), 69
 lapply(), 147
 layout(), 565
 layout.show(), 565
 lbl_test(), 374
 lcm(), 565
 lda(), 458
 legend(), 526
 length(), 20, 28, 117, 123
 LETTERS, letters, 28
 levels(), 55, 57, 58
 leveneTest(), 213
 library(), 17
 LillieTest(), 357
 lines(), 520
 list(), 117
 list.files(), 167
 lm(), 181, 186, 223, 245, 448, 450, 452
 lm.influence(), 199
 lm.ridge(), 205
 lmrob(), 203
 load(), 158
 loadhistory(), 13
 locator(), 517
 loess.smooth(), 547
 log(), log2(), log10(), 14
 logical, 23
 logLik(), 291
 loglin(), 319
 loglm(), 319
 logm(), 419
 lower.tri(), 78
 lowess(), 547
 irtest(), 302, 307
 ls(), 12, 22
 ltsReg(), 203
 mad(), 72
 mahalanobis(), 425
 Manova(), 456
 manova(), 449, 452, 455, 456
 mantelhaen.test(), 365
 mapply(), 150
 match(), 108
 matlines(), 520
 matplot(), 505
 matpoints(), 519
 matrix(), 75

mauchly.test(), **238**, 260, 266
 max(), 64
 max.col(), 303
 mcnemar.test(), 395, 396
 mean(), 65
 MeanAD(), 72
 median(), 65
 merge(), 138
 methods(), 582
 mh_test(), 398
 min(), 64
 missing(), 579
 mlv(), 66
 Mod(), 14
 mode(), 20, 23
 model.frame(), 182
 model.matrix(), 172, 182, **463**
 model.tables(), 222, 241
 monoMDS(), 446
 months(), 115
 mood.test(), 382
 mosaicplot(), 320, 514
 mtext(), 527

 n2mfrow(), 567
 NA, 14, 97
 na.fail(), 102
 na.omit(), **99**, 100, 102, 136
 names(), **32**, 119, 127
 NaN, 14
 nchar(), 104
 ncol(), 76, 123
 next, 576
 nlevels(), 55
 nls(), 208
 noquote(), 107
 norm(), 423
 nrow(), 76, 123
 ns(), 207
 NULL, 14
 Null(), 422
 numeric, 23
 numeric(), 27

 object.size(), 587
 odbc(Befehl)(), 161
 OddsRatio(), 368

 odTest(), 314
 on.exit(), 581
 oneway.test(), 220
 oneway.test(), 413
 options(), 11, 13, 102, 184, 467, 468, 580,
 585
 order(), **46**, 82, 131
 ordered(), 59
 outer(), 75

 PageTest(), 393
 pairs(), 561
 pairwise.t.test(), 229
 pairwise.wilcox.test(), 390
 palette(), 509
 par(), 507, 567
 parse(), 111
 PartCor(), 71
 paste(), 104
 pbinom(), 176
 pbirthday(), 175
 pchisq(), 176
 pcout(), 196
 pdf(), 501
 PearsonTest(), 359
 Permn(), 40, 417
 persp(), 552
 persp3d(), 553
 pf(), 176
 pi, 14
 pie(), 542
 pie3D(), 542
 plot(), 95, 198, 231, **502**
 plot.design(), 222, 241
 plot.new(), 517
 plot3d(), 553
 PlotFdist(), 535
 plotSlopes(), 192
 pmatch(), 108
 pmax(), 65
 pmin(), 65
 pnorm(), 176
 points(), 519
 polr(), 301
 poly(), 207
 polygon(), 523
 polypath(), 523

POSIXct, 113
 POSIXlt, 114
 PostHocTest(), 228, 252
 power.anova.test(), 283
 power.t.test(), 281
 powerTransform(), 201
 ppoints(), 541
 prcomp(), 434
 predict(), 194, 293, 303, 308, 344, 459
 princomp(), 434, 438
 print(), 21
 prod(), 64
 prop.table(), 89
 prop.test(), 363
 prop.trend.test(), 364
 pt(), 176

 q(), 10
 qbinom(), 177
 qchisq(), 177
 qda(), 457
 qf(), 177
 Qn(), 73
 qnorm(), 177
 qperm(), 414
 qqline(), 541
 qqnorm(), 541
 qqplot(), 540
 qr(), 427, **430**
 qt(), 177
 quantile(), 68, 332
 quarters(), 115
 quartz(), 499

 r.test(), 180
 R.Version(), 7
 range(), 64
 rank(), 50
 rasterImage(), 532
 rbind(), 81, 137
 rbind_all(), 137
 rbinom(), 44
 rchisq(), 44
 rcorr(), 180
 Re(), 14
 read.dta(), 161
 read.spss(), 160

 read.table(), 155
 read.xport(), 161
 readLines(), 165
 recode(), 52, 58
 rect(), 522
 regexpr(), 109
 relevel(), 60
 RelRisk(), 369
 remove.packages(), 17
 reorder(), 60
 reorder.factor(), 60
 rep(), 43
 repeat, 577
 replace(), 51
 replicate(), 148
 require(), 17
 reshape(), 144
 residualPlots(), 200
 residuals(), 182, 291, 341
 return(), 581
 rev(), 45
 rf(), 44
 rgb(), 510
 rgb2 hsv(), 510
 rle(), 90
 rlm(), 203
 rm(), 22
 rmvnorm(), 310, 424
 RNGkind(), 41
 rnorm(), 44
 roc(), 369
 round(), 13
 row(), 77
 rowMeans(), 83
 rownames(), 128
 rowSums(), 83
 Rprofile.site, 11
 rstandard(), 199
 rstudent(), 199
 rt(), 44
 rug(), 535
 runif(), 44
 RunsTest(), 354

 sample(), 43
 sandwich(), 204
 sapply(), 148

sasxport.get(), 161
 save(), 158
 save.image(), 158
 savehistory(), 13
 scale(), 49
 scaleTau2(), 73
 scan(), 157, 165
 scatter3d(), 187
 ScheffeTest(), 228, 252
 screen(), 569
 sd(), 69
 search(), 12
 segments(), 521
 select(), 205
 seq(), 42
 sessionInfo(), 7, 18
 set.seed(), 41
 setdiff(), 38
 setequal(), 38
 setRepositories(), 17
 setwd(), 11
 shapiro.test(), 357
 shell(), 7, 154
 showMethods(), 582, 584
 sign(), 13
 signif(), 13
 SignTest(), 386
 sim.<Typ>(), 123
 sin(), 14
 sink(), 7, 103
 Skew(), 69
 smooth(), 547
 smooth.spline(), 548
 smoothScatter(), 536, 546
 Sn(), 73
 sobel(), 183
 some(), 47
 SomersDelta(), 373
 sort(), 46
 source(), 153, 158
 spineplot(), 514
 spline(), 548
 splinefun(), 548
 split(), 136
 split.screen(), 569
 sprintf(), 105
 spss.get(), 160

sql<Befehl>(), 161
 sqrt(), 13
 sqrtm(), 429
 stack(), 141
 stars(), 551
 stata.get(), 161
 stem(), 537
 step(), 189
 stop(), 579
 stopifnot(), 579
 str(), 20, 124
 stripchart(), 539
 stripplot(), 560
 strptime(), 114
 StrRev(), 108
 strsplit(), 107
 structure(), 20
 sub(), 111
 subset(), 132
 substring(), 109
 sum(), 63
 summary(), 55, 63, 93, 123, 183, 198, 222,
 226, 238, 273, 295, 437, 455
 sunflowerplot(), 546, 551
 support(), 414
 supsmu(), 547
 Surv(), 326
 surv_test(), 333
 survdiff(), 333
 survexp(), 344
 survfit(), 331, 338
 survreg(), 347
 survSplit(), 329
 svd(), 429
 sweep(), 84
 switch(), 573
 symbols(), 551
 symmetry_test(), 394, 396, 397
 Sys.Date(), 112
 Sys.getlocale(), 46
 Sys.glob(), 168
 Sys.info(), 7
 Sys.setlocale(), 46
 Sys.time(), 113
 system.time(), 587
 T, *siehe* TRUE

t(), 418
 t.test(), 215, 217, 219
 table(), 88
 tail(), 124
 tan(), 14
 tapply(), 73
 terms(), 174
 testInteractions(), 247
 testSlopes(), 192
 text(), 527
 textConnection(), 157
 theme(), 556
 title(), 526
 tolower(), 107
 toString(), 103
 toupper(), 107
 traceback(), 585
 transform(), 130
 TRUE, 24
 trunc(), 13
 try(), 580
 tryCatch(), 580
 TschuprowT(), 373
 TukeyHSD(), 230, 253
 typeof(), 23
 unclass(), 55
 undebbug(), 586
 union(), 37
 unique(), 37, 135
 unlist(), 122
 unstack(), 142
 Untable(), 94
 update(), 188, 292
 update.packages(), 17
 upper.tri(), 78
 UseMethod(), 582
 var(), 69
 var.test(), 212
 vcov(), 182
 vcovHC(), 204, 313
 vector(), 27
 vglm(), 299, 306, 310, 312, 313, 315, 316
 View(), 124
 vif(), 202
 vignette(), 18

vss(), 443
 vuong(), 316
 warning(), 579
 weekdays(), 115
 weighted.mean(), 65
 which(), 36, 81
 which.max(), 64
 which.min(), 64
 while(), 576
 wilcox.test(), 387, 389, 390
 windows(), 499
 Winsorize(), 67, 72
 with(), 128
 write.foreign(), 160, 161
 write.table(), 157
 writeLines(), 166
 x11(), 499
 xlab(), 556
 xlim(), 556
 xor(), 24
 xspline(), 548
 xtabs(), 92
 xyplot(), 560
 ylab(), 556
 ylim(), 556
 YuleQ(), 369
 zapsmall(), 26
 zeroinfl(), 315

Zusatzpakete

abind, 87
adimpro, 532, 534
AICcmodavg, 185
Amelia II, 103
boot, 401, 493

Cairo, 501
car, 8, 47, 52, 58, 187, 196, 198, 200–202,
213, 237, 259, 265, 268, 269, 271,
455, 456, 529, 544
caret, 493, 497
coin, 333, 374, 394, 396–398, 413
colorspace, 510
compiler, 589
coxphf, 345

DAAG, 122
data.table, 123, 158, 588
DBI, 161
DescTools, 37, 40, 60, 63, 65, 67, 69–72, 94,
108, 180, 223, 228, 235, 242, 252,
257, 263, 352, 354, 357, 359, 368,
369, 373, 374, 376, 377, 379, 380,
386, 392, 393, 417, 449, 451, 530,
535
devtools, 17
dplyr, 130, 137, 147

EBImage, 532, 534
effects, 276
expm, 419, 429

foreign, 160

geepack, 208
ggplot2, 499, 553
ggviz, 553
glmnet, 205, 345
googleVis, 553
GPArotation, 441

hexbin, 546

Hmisc, 160, 161, 180, 518, 535
HSAUR2, 122

ipred, 493, 497

JGR, 5

lattice, 499, 560
lavaan, 439
leaps, 189
lme4, 208
lmtest, 200, 204, 313, 316
logistf, 298
lubridate, 112

MASS, 203, 205, 301, 313, 319, 422, 457, 458,
547
Matrix, 418
MBESS, 216, 278
mediation, 183
mgcv, 208
mice, 103
mi, 103
microbenchmark, 587
modeest, 66
multcomp, 225, 249, 266, 276
multilevel, 183, 374
mvoutlier, 196
mvtnorm, 175, 310, 424, 551

nlme, 208

OpenMx, 439

pcaPP, 434
perturb, 203
phia, 247
playwith, 553
plotrix, 511, 529, 542
polycor, 71
pracma, 419
pROC, 369

pscl, 314
psych, 123, 180, 441, 443
pwr, 278

rCharts, 553
RColorBrewer, 509
resample, 401, 413
reshape2, 144
rggobi, 553
rgl, 553
rms, 73, 291
robustbase, 68, 73, 203
rockchalk, 192
RODBC, 161
RSQLite, 163

sandwich, 204, 313
sem, 439
sets, 37
sos, 16
splines, 207
stringr, 103
survival, 288, 324, 326, 330, 331, 333, 335,
 339, 344, 347

timeDate, 112
tourr, 550

vcd, 288
vegan, 413, 446
VGAM, 299, 302, 305–307, 310, 312, 313, 315,
 316

XLConnect, 159